
SKY
CREATED BY TEW

Presentado por:

Mateo Hernandez

Ing. Sistemas y Comp.

SKY

GAME ENGINE AND GAME

Bienvenidos al proyecto Sky, este proyecto está basado en la mecánica del famoso videojuego “Flappy Bird” donde el objetivo principal es dirigir un “Pájaro” por medio de saltos a través de obstáculos (Pipes) que aparecen aleatoriamente enfrente del mismo, a diferencia de este juego, Sky se presenta en el espacio donde nuestra única meta será esquivar las diferentes estructuras alienígenas que deberemos enfrentar a velocidades inimaginables. A continuación, se podrá leer las funcionalidades de cada uno de los archivos que funcionan directamente en el juego (debido a que hay algunas funciones que tuvieron que ser retiradas).

NOTAS

Decidí entregar en el código final la IA, aunque no funcione por el hecho de que la propuesta inicial del proyecto era esa, por esto, algunos atributos están en protected aunque aquí en la explicación aparezcan como private. Todas las clases que se verán están ubicadas en un solo namespace llamado “Sky”. La presentación esta en orden de importancia de cada archivo, cada función podrá ver lo que hace, es recomendado tener el código fuente para entender por completo como funciona todo el programa.

VERSIÓN FINAL DE ENTREGA 3.6

GAME ENGINE

Class: Sky Control

//StateControl.h

Esta clase funciona para crear una nueva declaración en un puntero único que nos ayudara a crear y controlar los estados que ocurren dentro del juego. Utilizando funciones virtuales puras. Es la base para hacer utilización de sus métodos e implementarlos en cada estado.

Funciones Públicas:

```
virtual void Init() = 0; //Inicializar variables (estado)

virtual void HandleInput() = 0; //Recibir entrada de mouse y teclado (teclado principalmente)

virtual void Update(float dt) = 0; //Actualizar parámetros teniendo en cuenta la variable de entrada dt (delta time) que contiene el espacio entre fotogramas

virtual void Draw(float dt) = 0; //Dibujar las texturas que se mostraran en la ventana del juego.

virtual void Pause() { } //Referencia a pausar un estado


virtual void Resume() { } //Referencia a resumir un estado

No tiene atributos.
```

Class: StateMachine

//StateMachine.h | StateMachine.cpp

Controla el procesamiento de movimiento entre estados, método para añadir un nuevo estado, eliminarlo y obtener el estado activo.

```
typedef std::unique_ptr<SkyControl> StateRef; Asignacion de la declaración de puntero único como StateRef. Referencia a los estados.
```

Funciones Públicas:

```
StateMachine() { } //Constructor

~StateMachine() { } //Destructor

void AddState(StateRef newState, bool isReplacing = true){
//Metodo para añadir nuevo estado tomando como parámetros el nuevo estado y la variable booleana de si está reemplazando o no. Debe inicializar la variable "_isAdding" a
```

```
verdadero, "_isReplacing" haciendo referencia y transferir los datos del nuevo estado que
ingresa.
}

void RemoveState(){
//Poner la variable booleana "_isRemoving" a verdadero
}

void ProcessStateChanges(){
//Encargada de procesar los movimientos en los estados, debe de chequear si el stack de
"_states" esta vacio y si esta removiendo para quitar el estado que esta actualmente en
la "punta" funcionando, al igual si esta añadiendo para pausar el estado que esta
funcionando y empujar el nuevo estado a la cima del stack.
}

StateRef& GetActiveState(){

//Obtener que estado está en funcionamiento actualmente.
}

Atributos Privados:

std::stack<StateRef> _states; //Stack de estados

StateRef _newState; //Puntero único de SkyControl

bool _isRemoving, _isAdding, _isReplacing; //Variables de control para proceso de cambio
de estados.
```

Class: AssetManager

//StateMachine.h | StateMachine.cpp

El assetmanager es el encargado de añadir las texturas y fuentes al juego, cargarlas.

Funciones Públicas:

```
AssetManager() { } //Contructor
~AssetManager() { } //Destructor

void LoadTexture(std::string name, std::string fileName){
//Metodo que carga las texturas tomando como parametros el nombre de la textura "name" y
su directorio "fileName" crea una textura y la guarda en el map "_textures"

sf::Texture& GetTexture(std::string name){
//Obtener la textura, retorna la misma textura de acuerdo al nombre que se le pase "name"
}

void LoadFont(std::string name, std::string fileName){
//Funciona de la misma manera que "LoadTextura" pero importando fuentes
}
```

```
sf::Font& GetFont(std::string name){

//Funciona igual que "GetTexture" pero obtiene la fuente

}
```

No tiene atributos.

Class: InputManager

//InputManager.h | InputManager.cpp

Encargada de recibir las entradas de Mouse y teclado, sin embargo, solo esta optimizada la entrada de teclado (ESPACIO).

Funciones Públicas:

```
InputManager() {} Constructor
~InputManager() {} Destructor

bool IsSpriteClicked(sf::Sprite object, sf::Mouse::Button button, sf::RenderWindow&
window){
//Permite saber en qué momento está siendo "clickeada" la textura (Sprite) que se pase en
el parámetro "object", se obtiene que botón del mouse se esta presionando en "Button" y
la ventana donde se está ejecutando todo (window). }

bool IsSpriteClicked2(sf::Keyboard::Key button){//Solo se encarga de obtener la entrada
de teclado con el parámetro "Button"}

sf::Vector2i GetMousePosition(sf::RenderWindow& window){

//Se encarga de devolver un vector de coordenadas (específicamente enteras) x y y,
pasando como parámetro la ventana donde se ejecuta

}
```

No tiene atributos.

GAME

Class: Game

//GameControl.h | GameControl.cpp

Encargada de controlar el loop que se necesita mientras la ventana esta abierta para actualizar los valores y controlar los fotogramas que se muestran por segundo, también contiene una función para añadir estados.

```
struct GameData{
//Contiene cada uno de los elementos del game engine mas una ventana (sf::RenderWindow)
}

typedef std::shared_ptr<GameData> GameDataRef{
//Se crea una nueva declaración para vector compartido de tipo estructura “GameData” y se asigna GameDataRef haciendo referencia a toda la data del juego
}
```

Funciones Públicas:

```
Game(int width, int height, std::string title){
//Constructor de la clase game que obtiene los parámetros de ancho y alto (width and height) para definir el tamaño de la ventana de juego y el la variable que define el titulo de la ventana que se mostrara en la parte superior: “title”
}
```

Atributos Privados:

```
const float dt = 1.0f / 60.0f; //Delta time, espacio entre fotogramas
sf::Clock _clock; //Reloj para medir el tiempo dentro durante la ejecución del juego

GameDataRef _data = std::make_shared<GameData>(); //Puntero compartido que contiene el acceso a todos los elementos que conforman el game engine


void Run(); {
//Funcion privada que se declara en el constructor y ejecuta el loop del juego mientras la ventana se encuentre abierta, controla los fotogramas para que no superen el limite de 0.25 por segundo y realiza una interpolación para obtener el valor de delta que se necesita para dibujar las texturas en pantalla
}
```

STATES

Class: SplashState : SkyControl

//SplashState.h | SplashState.cpp

El Splash State es el mas sencillo de todos, solo se encarga de mostrar una imagen durante tres (3) segundos (variable definida en DefinitionsControl.h), pasa al top del stack y luego es reemplazado con el menú principal (MainMenuState.h), es hija de SkyControl.

Funciones Públicas:

```
SplashState(GameDataRef data){
//Constructor del SplashState que se le pasa el parámetro puntero de “data” con los componentes del game engine. Aquí no se inicializan las variables, solo el constructor.
}

void Init(){
//Inicialización de las texturas de acuerdo a su nombre y la variable con el contenedor de su directorio, se le dice a la variable “_background” que tome el valor de la textura importada.
}

void HandleInput(){
//En este metodo se crea un evento donde se rectifica si el evento se cierra al igual debe hacerlo la ventana, para manejar el input por parte del usuario y decida cerrar la ventana
}

void Update(float dt){
//Se realiza la utilizacion de un reloj para obtener el tiempo y compararlo con la variable definida en “DefinitionsControl.h” “SPLASH_STATE_SHOW_TIME” que es el tiempo que se demora el estado en desaparecer de la pantalla y dejar pasar al nuevo estado que ingresa (MainMenuState). No hace uso de delta time (dt).
}

void Draw(float dt){

//Contiene solo tres (3) comandos, cambiar el fondo a rojo (en caso de que la textura no llene la pantalla y sea mas fácil identificarla), dibujar el fondo (que es la textura que importamos al principio) y el comando “.display” que muestra en pantalla todo lo que se ha cargado.}
```

Atributos Privados:

```
GameDataRef _data; //Puntero para la inicialización y utilización del game engine

sf::Clock _clock; //Reloj para controlar

sf::Sprite _background; //Textura que guarda el fondo del Splash State
```

Class: MainMenuState : SkyControl

//MainMenuState.h | MainMenuState.cpp

El estado de menú principal está compuesto por una textura de fondo y un array tipo vector que contiene los fotogramas de la textura que al darle "click" te lleva al juego directamente. Es hija de SkyControl.

Funciones Públicas:

```
MainMenuState(GameDataRef data){
//Este constructor solo se encarga de inicializar SkyControl y la variable
"_animationrunnerPlay" para hacer la animación del título de inicio (textura).
}

void Init(){
//En este método se cargan todas las texturas de acuerdo a su nombre y directorio, de
igual manera como se hizo en el Splash State, en especial las del titulo principal que se
aseguran en un vector para hacer la animación. Se asigna un fondo con la variable
"_background" y se ubica en su posición: La mitad de su ancho divido entre dos para X, el
alto de la pantalla entre dos menos el tamaño de la altura del botón entre dos:

_playButton.setPosition((SCREEN_WIDTH / 2) - (_playButton.getGlobalBounds().width / 2),
(SCREEN_HEIGHT / 2) - (_playButton.getGlobalBounds().height / 2));
}

void HandleInput(){
//Funciona de la misma manera que en el Splash State con la diferencia que se hace un
chequeo de si la textura del titulo principal esta siendo "clickeada" por el mouse con la
función que declaramos en el InputManager "IsSpriteClicked" y se realiza el paso al Game
State para empezar a jugar
}

void AnimationPlay(float dt){
//Encargado de correr la animación del botón de inicio (el titulo), utilizando la
variable "_animationrunnerPlay" que recorre el vector contenedor de fotogramas y cuando
llega al final vuelve al inicio, cambia la textura de "_playButton" de acuerdo a donde
este "_animationrunner" y da el efecto de que se esta moviendo. Al finalizar receta el
reloj para controlar el proceso con una variable que se define en "DefinitionsControl.h"
}


void Update(float dt){
//Implementa el método AnimationPlay constantemente mientras el estado este abierto
}
void Draw(float dt){

//De igual manera que los demás estados este método muestra todas las texturas en
pantalla.

}
```

Atributos Privados:

```
GameDataRef _data; //Puntero para la inicialización y utilización del game engine
```

```
sf::Sprite _background; //Textura del fondo
sf::Sprite _title; //Textura del titulo (no utilizada)
sf::Sprite _playButton; //Textura del botón para empezar a jugar

std::vector<sf::Texture> _animationFramesPlay; //Vector contenedor de las texturas del
botón de play
sf::Clock _clock; //Reloj para la animacion
int _animationrunnerPlay; //Control del recorrido de los fotogramas y la animación
```

Class: GameState : SkyControl

//GameState.h | GameState.cpp

El game state es donde ocurre la mayoría de eventos que el jugador experimenta, porque ya es el juego en si, se importan todas las texturas que componen el fondo, la nave y los obstáculos, al igual que el "score" que aumenta conforme al la nave pasa mas obstáculos. Es hija de SkyControl.

Funciones Públicas:

```
GameState(GameDataRef data){
//Solamente encargada de inicializar SkyControl
}

void Init(){
//En este método se importan todas las texturas del personaje, los obstáculos, el piso,
la fuente del puntaje, se crean nuevos objetos a partir de los punteros que se declaran
en los atributos, se inicializa el fondo con su textura, el score a cero y se establece
el estado actual del juego a "Start"
}

void HandleInput(){
//Funciona de igual manera que el estado "MainMenuState" a diferencia que al hacer
"click" en la textura de fondo o presionando la tecla "ESPACIO" se activa la función de
la nave que le permite moverse en el eje Y y hacer el efecto de que esta subiendo y
bajando cuando se deja presionar la tecla.
}



void Update(float dt){
//Este, en sí, es el método mas robusto de todos: controla el movimiento de todo en
pantalla cuando el juego inicia, hace un chequeo de colisiones para saber si en algún
momento el personaje "se estrella" con alguno de los obstáculos o el "piso" se dejen de
mover los elementos simplemente dejando de llamarlos y estableciendo el estado actual del
juego a "Over". Se hace la aparición de los obstáculos, la aleatoriedad y los invisibles
para el score. Por último en caso de pasar el estado del juego a "Over" se llama a un
nuevo estado con la misma data y se "resetea" todo.
}

void Draw(float dt){
```

```
//Funciona de la misma manera que los demás métodos Draw() solo dibujando todo en
pantalla, en este caso cada uno de los objetos.

}

Atributos Privados:

GameDataRef _data; //Puntero para la inicialización y utilización del game engine

sf::Sprite _background; //Textura del fondo

Obstacles* Ob; //Puntero para hacer llamado al objeto “Obstacles” (obstáculos)

SpaceRocks* sR; //Puntero para hacer llamado al objeto “SpaceRocks” (el piso)

StarShip* STS; //Puntero para hacer llamado al objeto “StarShip” (nave)

Collisions collision; //Llamado a los métodos de colisiones

Score *_score; //Puntero para tomar el score del personaje

sf::Clock clock; //Reloj para controlar el tiempo del juego

int _gameActualState; //Variable entera que se controla con la enumeración generada en
“DefinitionsControl.h”

int numberDetectionScore; //Variable que actualiza el score del personaje (puntaje en si)
```

OBJETOS

Class: SpaceRocks

//SpaceRocks.h | SpaceRocks.cpp

Esta clase no es más que el planeta donde volara la nave, está hecho de una textura que se mueve de acuerdo al movimiento de los obstáculos (para que todos vayan a la misma velocidad) en un vector que elimina la primera textura cada que se sale de la pantalla por complete, esto hace el efecto de que el planeta es infinito.

Funciones Públicas:

```
SpaceRocks(GameDataRef data){
//Constructor encargado de obtener las texturas del “piso” que se van a utilizar, definir su posición en la parte inferior de la pantalla y enviarlas dentro del vector de texturas
}

void MoveRocks(float dt){
//Utilizando un ciclo que recorra el vector de texturas definimos una variable (movement) que toma el valor del movimiento de obstáculos multiplicado por delta time (para que se muevan de acuerdo a los fotogramas y no haya errores de movimiento); en este método se elimina la textura que se sale de la pantalla y se pasa al principio. Todo esto controlado por un vector de coordenadas flotantes que se obtiene por el tamaño de la pantalla y la posición de las texturas en el eje Y.
}

void DrawRocks(){
//Como todos los métodos Draw() este recorre el vector de texturas y dibuja en pantalla utilizando el game engine: _data->window.draw(_spaceRocks.at(i));
}

const std::vector<sf::Sprite>& GetSprites() const{

//Esta función retorna el vector “_spaceRocks” contenedor de las texturas del “piso”

}
```

Atributos Privados:

```
GameDataRef _data; //Puntero para la inicialización y utilización del game engine

std::vector<sf::Sprite>_spaceRocks; //Vector de texturas para hacer el efecto de movimiento y mostrar el “piso”
```

Class: StarShip

//StarShip.h | StarShip.cpp

La Star Ship, esta es la nave y por supuesto el personaje que va a controlar el usuario, se mueve de igual forma que las demás animaciones que se realizan dentro de la ventana (un vector), cuenta con un método en especifico que le permite moverse en el eje Y manteniendo su posición en el X.

Funciones Públicas:

```
StarShip(GameDataRef data){
//En este constructor se obtienen todas las texturas y se envían directo al
“_anumationFrames” que es un vector con todas las texturas para hacer la animación de la
nave. Se define su posición inicial en pantalla.
}

void Draw(){
//Se dibuja la starship usando el game engine
}

void Animation(float dt){
//Se realiza el mismo procedimiento que se realizo cuando se estaba animando el botón del
menu principal (MainMenuState)
}

void Explosion(){
//Se define la textura de la nave a uan textura aparte que simula la explosión de la
misma
}

void Update(float dt){
//Este método en versiones anteriores realizaba un pequeño salto en la nave siempre que
se presionaba “ESPACIO” o algún botón del mouse, pero en esta última versión su única
tarea es aumentar el movimiento de la nave en Y cuando se presione la tecla y el estado
de la nave sea: “STARSHIP_STATE_FLYING” utilizando una ecuación sencilla que disminuye y
controla la velocidad; en caso de que este “STARSHIP_STATE_FALLING” se dejara caer la
nave por una variable de “gravedad.”
}

void Up(){
//Cambiar el estado de la nave a “STARSHIP_STATE_FLYING”
}

const sf::Sprite& GetSprite() const{

//Retorna el vector de sprites (texturas) de la Star Ship

}
```

Atributos Privados:

```
GameDataRef _data; //Puntero para la inicialización y utilización del game engine

sf::Sprite _starship; //Textura principal de la starship
```

```
std::vector<sf::Texture> _animationFrames; //Vector de texturas de la Starship

sf::Texture _FIRE; //Textura especial para hacer la explosión de la nave

unsigned int _animationrunner; //Variable que solo toma valore positivos (controla la
animación de la nave)

sf::Clock _clock; // Reloj para la animación
```

Class: Obstacles

//Obstacles.h | Obstacles.cpp

Esta clase se encarga solamente de los obstáculos que se mueven de acuerdo a una variable definida en "DefinitionsControl.h" y van apareciendo de manera aleatoria, están contenidos en un vector con todas sus texturas y tienen una textura invisible que detecta la posición del jugador para aumentar el puntaje con la clase "Score".

Funciones Públicas:

```
Obstacles(GameDataRef data){
//El constructor inicializa la variable "_spaceheight" que obtiene el alto de las
texturas del "piso", la variable "_rocksSpawnY" a cero y una "seed" para controlar los
números aleatorios
}

void SpawnBottomOb(){
//Su función es obtener la textura del obstáculo de la parte superior y ubicarlo en su
posición (a esta posición se le aumento manualmente un tamaño de 70 pixeles para
acomodarlos bien en pantalla) e inmediatamente se envían al vector de obstáculos
(_obstacles)
}

void SpawnTopOb(){
//A diferencia del "SpawnBottonOp", este aparece los obstaculos de la parte inferior
realizando el mismo proceso.
}

void SpawnInvisibleOb(){
//Importa una textura transparente (.png vacío) de un tamaño un poco mas grande que la
pantalla para cubrir y detectar cuando la nave pasa a través de los obstaculos. Tambien
se envia a el vector de obstaculos.
}

void MoveOb(float dt){
//Se realiza un recorrido por el vector de obstaculos moviendo cada uno de ellos en el
eje X y eliminando los que salen de la pantalla (es decir, rectificando que su posición
en x sea menor a cero), al igual se hace un recorrido por el vector de obstáculos
invisibles para hacer mas luego el "Score".
}
```

```
void DrawObs(){
//Dibuja todos los obstáculos que se encuentran en el vector “_obstacles”
}
void RandomizeObstacles(){
//Asigna a la variable “_rocksSpawnY” un valor aleatorio con la función rand() para que
los obstáculos no aparezcan siempre en la misma posición.
}

void SpawnPassDetection(){
//Funciona de igual forma que los otros métodos de “Spawn” pero con las texturas
“invisibles”
}

const std::vector<sf::Sprite>& GetSprites() const{
//Retorna el vector de texturas de “_obstacles”
}

std::vector<sf::Sprite>& DetectionObstacles(){

//Retorna el vector de texturas invisibles (_detectionnumber)

}
```

Atributos Privados:

```
GameDataRef _data; //Puntero para la inicialización y utilización del game engine

std::vector<sf::Sprite> _obstacles; //Vector de texturas de los obstaculos
std::vector<sf::Sprite> _detectionnumber; //Vector de texturas “invisibles”

int _spaceheight, _rocksSpawnY; //Variables para establecer la posición de los obstáculos
y el numero aleatorio
```

ELEMENTS

Class: Collisions

//Collisions.h | Collisions.cpp

Esta clase es parte del game engine, sin embargo, decidí ponerla de ultimo porque es un sistema muy sencillo que detecta la colisión entre dos elementos (texturas) que se mueven a través de la pantalla.

Funciones Públicas:

```
Collisions(); Constructor

bool checkIfColliding(sf::Sprite sprite1, float scale1, sf::Sprite sprite2, float
scale2){

//Se pasan como parámetros dos texturas (Sprite 1 y Sprite 2), se les asigna una escala
para eliminar los espacios en "blanco" que tienen los png y hacer la colisión mas
"realista", una vez ambas se tocan entre si con las escalas se envía verdadero, en caso
contrario: falso:

if (rect1.intersects(rect2)){
	return true;

}else{

	return false;
}

}
```

Class: Score

//Score.h | Score.cpp

La clase score solo controla el texto que aparece en la parte superior izquierda que se actualiza constantemente para hacer el efecto de que se aumenta el "puntaje" cada que el jugador va más lejos.

Funciones Públicas:

```
Score(GameDataRef data){
//Constructor que inicializa el game engine, define el "string" importándolo desde su
nombre que hace referencia al directorio, lo inicializa en cero, define un tamaño, color
y su posicion inicial en la pantalla.
}
~Score(); Destructor
```

```
void Draw(){
//Dibujar en pantalla la fuente
}

void Update(int score){

//Actualizar la fuente convirtiendo la variable entera “score” en string

}

Atributos Privados:

GameDataRef _data; //Puntero para el game engine

sf::Text _scoreText; //Fuente
```

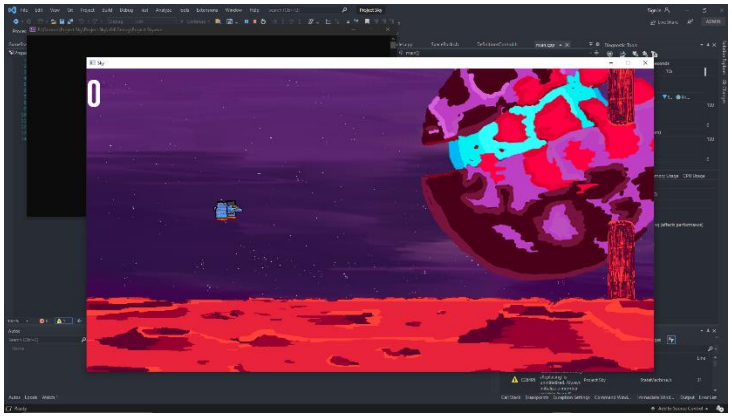
MAIN AND DEFINITIONS

En el archivo main.cpp solo se encuentra una función y la declaración de ciertas variables que funcionan para un modo especial que se activa desactivando unos comentarios en el “GameState.h” y el archivo “DefinitiosControl.h” que es donde están todas las definiciones de los directorios y las variables globales que se importan a lo largo de todo el programa.

Main.cpp inicio de juego

```
Sky::Game(SCREEN_WIDTH, SCREEN_HEIGHT, "Sky"){
//Se pasa el ancho y alto de la pantalla y se le asigna el nombre
}

Al correr esto:
```



El juego estará listo para jugar.