VIETNAMESE GERMAN UNIVERSITY

-------ᘛ📖ᘚ-------

C++ Project

"Doodle Jump"

May 2018

1. **Nguyen Phan Bao Viet - 11239**
2. **Nguyen Tran Minh Luan - 11467**

Instructor: Dr. Tran Phuong Nga, PhD

Abstract

This paper is about a C++ game project with respect to object-oriented programming. To begin, we have learned many things of C++ programming language at the beginning and then developing the game by what we learned. The report mentions the progress of work, and code explanation as the main parts and others which we will include later.

1. Introduction

Doodle Jump is a platforming video game developed and published by Lima Sky, for Windows Phone, iOS, BlackBerry, Android, Java Mobile (J2ME), Nokia Symbian, and Xbox 360 for the Kinect platform. It was released worldwide for iOS on March 15, 2009 and since its release, the game has been generally well received.

In the summer semester of the first year at VGU, we have joined the course "High-level programming language" for 4 weeks. In this course, we have to do an exam and a group project requiring us to build program which is used OOP in C++ language and eventually we decided to build "Doodle Jump"

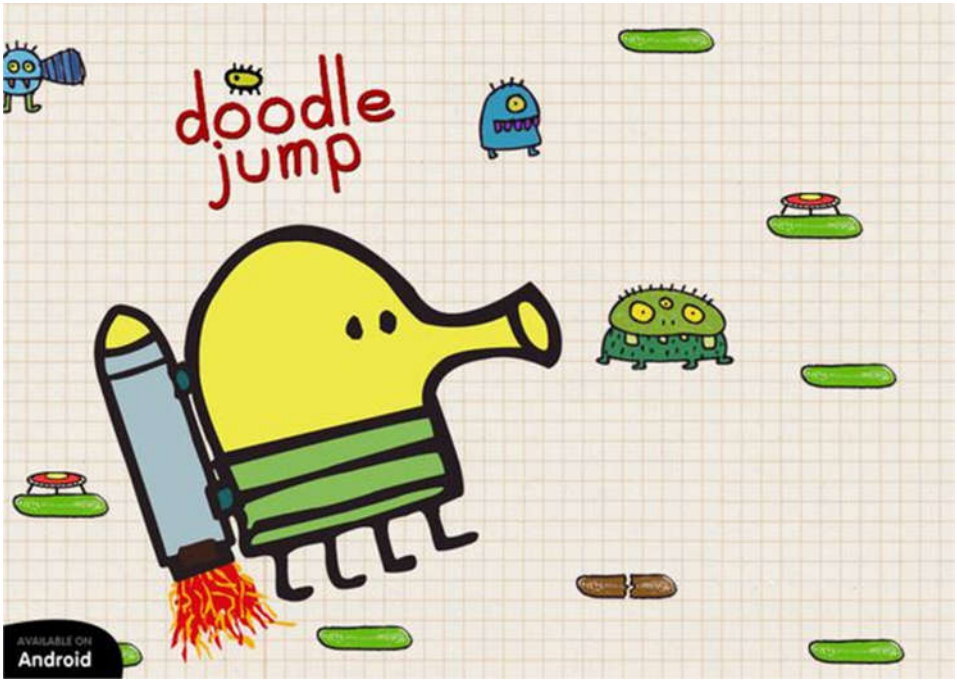


Figure 1. The game Doodle Jump

2. Background

In the summer semester of the first year at VGU, we have joined the course "High-level programming language" for 4 weeks. In this course, we have to do an exam and a group project requires us to build program which is used OOP in C++ language. As we all know, C++ is a very popular programming language that almost developers and programmers learned, from basis to

master. In detail, we have been learning OOP which is a programming paradigm of C++ in this course and must apply what we learned in this project.

3. Description of the work

In this project, we program Doodle Jump using the library Allegro5.2.4 in Visual Studio. The main idea of this game is that the player using the buttons on the keyboard to control a character called “Doodle” jump on platforms. When the game starts, Doodle will jump up and down continuously and he jumps up when his legs meet the platform below until reaching a certain height and he jumps down until he meets the old platform or another one, if he meets nothing and drop to the bottom side of the display, the game is over. The player controls Doodle to move left or right together with jumping up and down to keep him alive, it is mean that there must be a platform below Doodle at least. When Doodle jumps to a higher height, the score also increases, it is clear that he must jump to a new platform above the old one to let the score go up and when he jumps to a new platform, the display of platform will go down, it means the old platform which lower than his platform now will go down until passing the bottom limit of the display and then it disappear; at that time, the new platforms are also generated at the top limit of the display randomly.

The project will mainly focus on learning how to program using C++ in object-oriented programming. Here we build our game, without copy any sample given source code but we still borrow some logical ideas and the graphics.

4. Game Play

Firstly, when the game is released, the display will be showed like this:

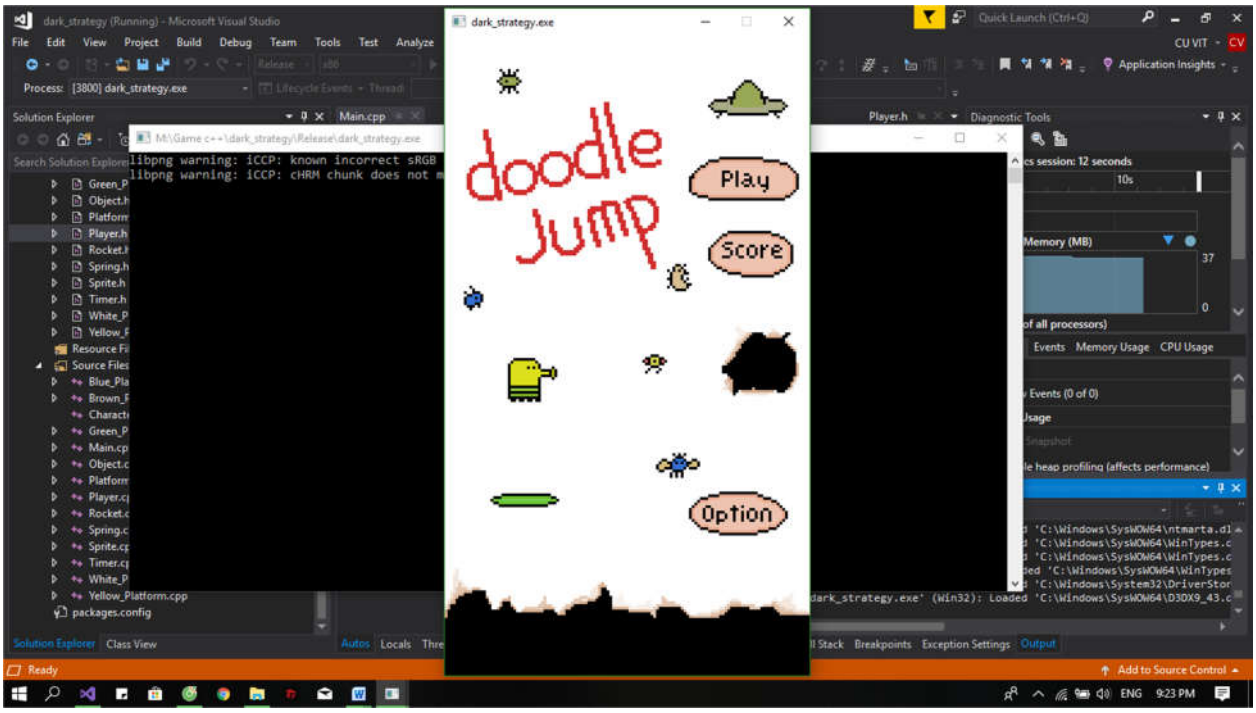


Figure 2. The menu of Doodle Jump

At this menu, the player will use mouse to add click on the command:

Play: to play the game
Score: to see the high score
Option: to set some other thing(we still have done nothing for this)
After adding click on **Play**, the game is like this:

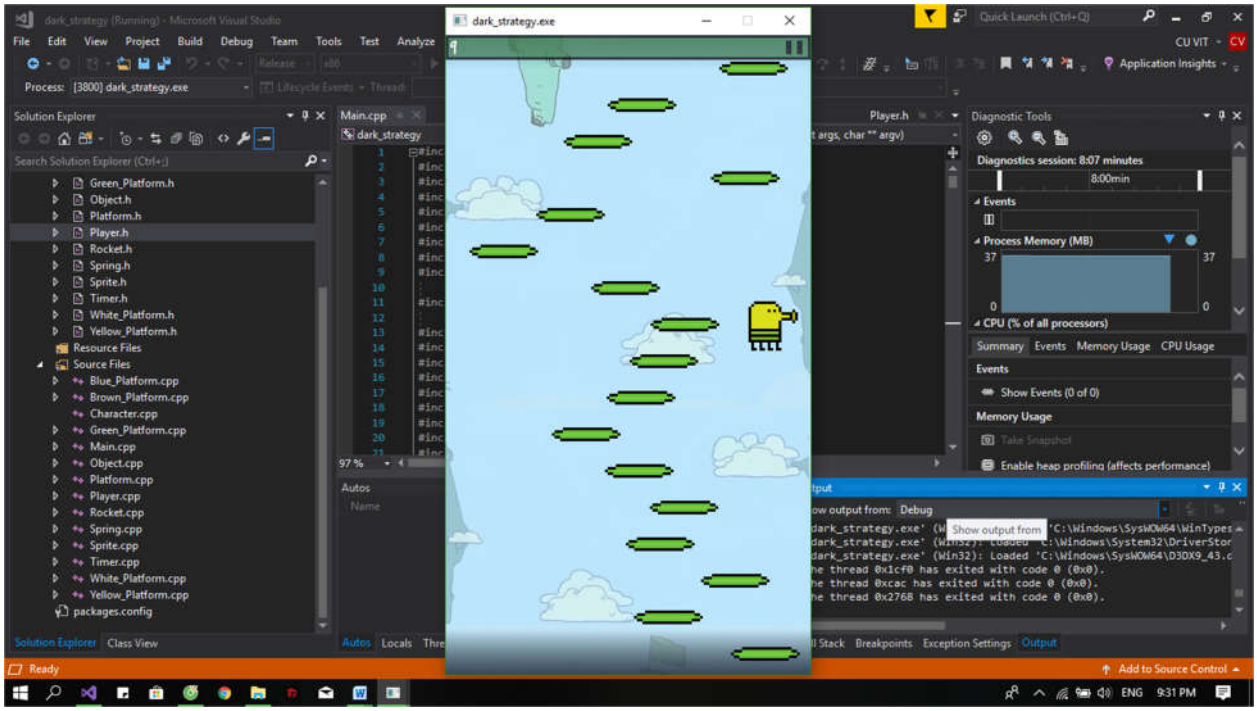


Figure 3. The beginning of Doodle Jump

When beginning, Doodle will randomly appear and below him must be a platform which can jump on.
To the left of bar located to the top of the display is score to see how good the player did. From early game, it is very easy to jump on platform but later, the rate of generating platform is decreased but still at a limit distance which allows Doodle can jump from below to the local one.
To controll Doodle, player just use the left and right arrows to control Doodle move left and right.
There are 5 types of platforms:

Green: This is basic platform, fixed postion and just used for jumping on
Blue: This platform can continuously move to left and right and used for jumping on
Yellow: This platform can be jumped on once, and then disappear.
White: This platform is special, it can appear too much but when being jumping on, all of the white platform which can be seen on display will randomly move along the horizontal line of each.
Brown: This platform can not be jumped on, if Doodle jumps on it, Doodle will not jump and keep falling and the brown platform will disappear.

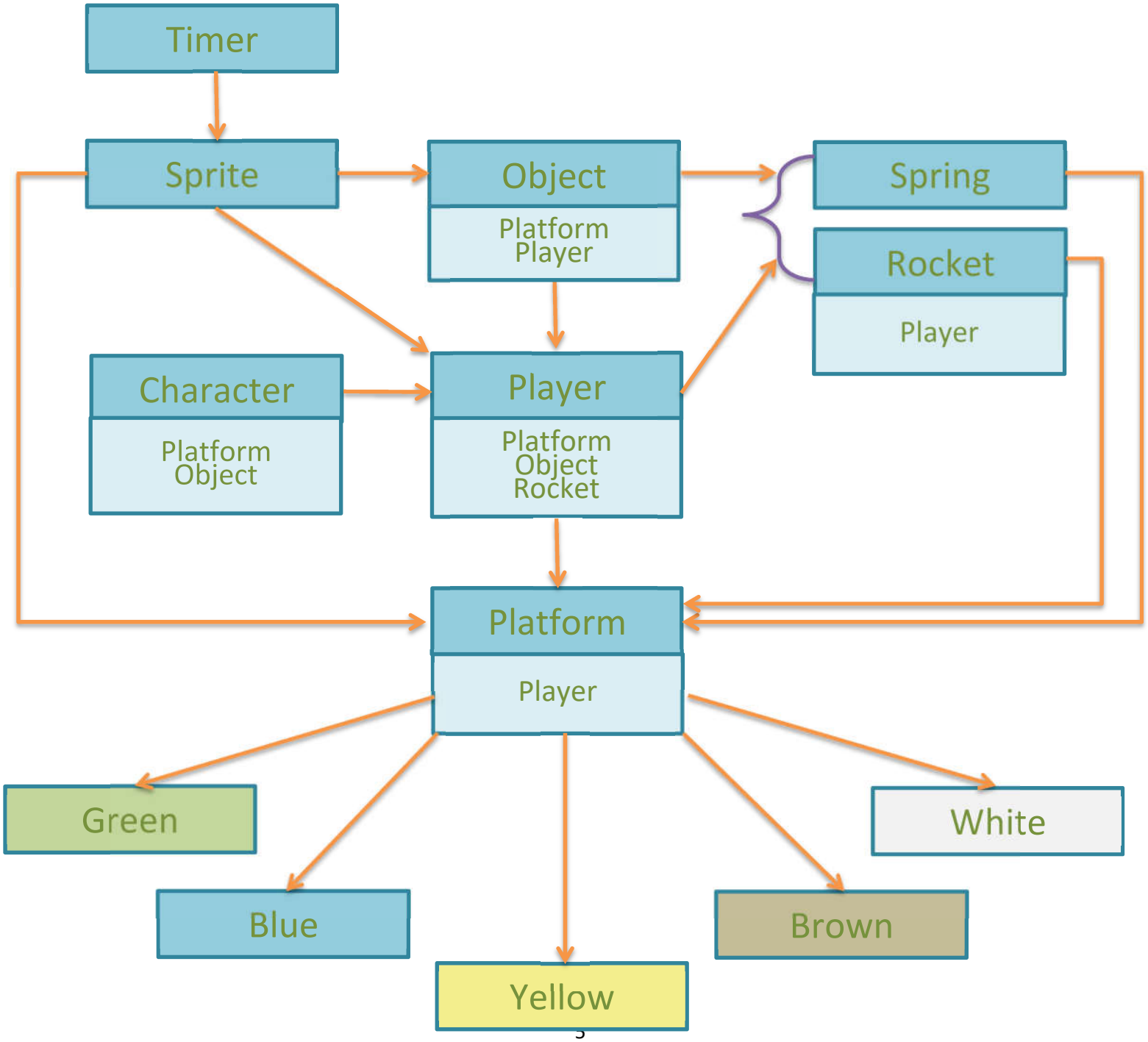
Besides, there are 2 other effects:

Spring: If doodle meets spring, doodle will fastly jump up and ignore other object.
Rocket: if doodle meets rocket, doodle will rocket up really really fast.

The game is over if Doodle drop directly to the bottom and does not attach or collide to any objects and platforms.

5. Code Explanation

A. Class Diagram



B. Class Explanation

First, we create the timer for the animation of object.

```
#include <allegro5\allegro.h>

class Timer
{
private:
	double checkpoint, max_time;
public:
	Timer(double max_time);
	~Timer();

	void reset();
	bool passed();
};

#include "Timer.h"


Timer::Timer(double max_time)
{
	checkpoint = al_get_time();

	this->max_time = max_time;
}
Timer::~Timer()
{
}

void Timer::reset() {
	checkpoint = al_get_time();
}

bool Timer::passed() {
	return al_get_time() - checkpoint > max_time;
}
```

We use command **al_get_time()** to get the local time now and the value is checkpoint to reset.
We use boolean to check whether the **passed()** happened or not and return the value we need, and **max_time** is actually the delay between images. This step is quite confused so we will explain more in the next part: Sprite.

Sprite

```
#include <allegro5\allegro.h>
#include "Timer.h"

class Sprite
{
private:
	ALLEGRO_BITMAP *image;
	Timer *timer;
	int current, quantity, width, height;
public:
	bool allow = true;
	Sprite(const char *filename, int quantity, double delay);
	Sprite(ALLEGRO_BITMAP *img, int quantity, double delay);
	~Sprite();

	void draw(float x, float y, int flag = 0, float tint_percent = 1);
	void next();

	int get_width();
	int get_height();
	int get_quantity();
	void get_current_of(Sprite *);
};
```

To begin, we have to say that our animation is loaded from images which are listed as a column like this.

So we create 4 variables at can be seen in the code. The **current** will check the local image, **quantity** is the quantity of the images(in sample is 4) **Width** and **height** are just width and height of the total image.

```
#include "Sprite.h"
#include <iostream>


Sprite::Sprite(const char *filename, int quantity, double delay)
{
	image = al_load_bitmap(filename);
	this->quantity = quantity;
```

```
	width = al_get_bitmap_width(image);
	height = al_get_bitmap_height(image) / quantity;
	timer = new Timer(delay);
}
```

At this, we create a constructor to load the image including the filename, quantity of images and delay. **Al_load_bitmap(filename)** will upload the image and gets the quantity of images. Now, **width** and **height/quantity** is the width and height of the image(we must devide by **quantity** because **al_get_bitmap_height(image)** will get the height of the large picture, and **timer** is create as delay(here is also the **max_time** before).

```
Sprite::Sprite(ALLEGRO_BITMAP *img, int quantity, double delay)
{
	image = img;
	this->quantity = quantity;

	width = al_get_bitmap_width(image);
	height = al_get_bitmap_height(image) / quantity;
	timer = new Timer(delay);
}
```

Next, we create another constructor to use the image which is loaded before. It's a little bit the same.

```
Sprite::~Sprite()
{
	delete timer;
	//al_destroy_bitmap(image);
}

void Sprite::draw(float x, float y, int flag, float tint_percent){
	al_draw_tinted_bitmap_region(image, al_map_rgba_f(1, 1, 1, 1 - tint_percent), 0,
height * current, width, height, x, y, flag);
	if (timer->passed() == true) {
		next();
		timer->reset();
	}
}
```

The function **draw** here is to draw the image we loaded(the image now is the lonely image we devided from the large picture). Now we check the condition by the timer whether it passed the delay or not, if passed we call function **next()** and reset timer.

```
void Sprite::next() {
	if (allow || current != quantity - 1) current = (current + 1) % quantity;
}
```

At this, if the boolean **allow**(we create at the header file) or the **current** image now is different from **quantity – 1**(because of couting from 0) so the **current** now

will get the value **(current +1)%quantity**. This can be considered to a loop to load image one by one and repeat the process.

```
int Sprite::get_height() {
	return height;
}

int Sprite::get_width() {
	return width;
}

int Sprite::get_quantity() {
	return quantity;
}
void Sprite::get_current_of(Sprite *other) {
	current = other->current;
}
```

These steps are to get the value we need.


```
#include "Character.h"
#include "Sprite.h"
#include "Object.h"

class Rocket;

class Player :
	public Character
{
private:
	Sprite *current, *jumping, *falling, *shooting;
	Object *attach;
	int flag = 0;
public:
	friend Platform;
	friend Object;
	friend Rocket;

	Player(const char * config_file, double delay);
	~Player();

	void show(float tint_percent, float max_width);
	void update(float gravity);

	int get_width();
	int get_height();
	float get_y();
	float get_x();
	int get_flag();
```

```
        bool is_falling();
        bool die(int max_height);

        void set_y(float y);
        void set_location(float x, float y);
        void set_dy(float dy);
        void set_dx(float dx);
        void set_jumping();
        void set_flag(int flag);

        float get_dx();
        float get_dy();
};
```

Firstly, this must be inherited from **Sprite.h** because of drawing using. About **Object.h** and **Character.h** as well as the friend class, we will explain later. Next we will clarify this class at CPP file.

```
#include "Player.h"
#include "Object.h"
#include <iostream>

unsigned convert(const char * str) {
        unsigned ans = 0;
        for (int idx = 0; str[idx] != '\0'; idx++) ans = ans * 10 + (str[idx] - '0');
        return ans;
}
```

We create a function called **convert** to get the quantity of the image

```
Player::Player(const char * config_file, double delay)
{
        ALLEGRO_CONFIG *cfg = al_load_config_file(config_file);

        attach = NULL;

        const char *jmp, *fll;
        jmp = al_get_config_value(cfg, "", "jumping");
        fll = al_get_config_value(cfg, "", "falling");

        int jmp_quann, fll_quann;

        jmp_quann = convert(al_get_config_value(cfg, "", "jumping quantity"));
        fll_quann = convert(al_get_config_value(cfg, "", "falling quantity"));

        jumping = new Sprite(jmp, jmp_quann, delay);
        falling = new Sprite(fll, fll_quann, delay);

        current = falling;
}
```

At this, we upload the image **jumping** and **falling** and then use **convert()** to get its quantity. This step can be skipped but if we want to create an animation for the player so we can apply this.

```
Player::~Player()
{
	delete jumping;
	delete falling;

	if (attach != NULL) delete attach;
}
```

In destruction we just delete all the sprite.

```
void Player::show(float tint_percent, float max_width) {
	if (x > max_width) x -= max_width;
	if (x + current->get_width() < 0) x += max_width;
	if (attach != NULL) {
		if (attach->x > max_width) attach->x -= max_width;
		if (attach->x + attach->get_width() < 0) attach->x += max_width;
	}
	current->draw(x, y, flag, tint_percent);
	if (attach != NULL) attach->show(flag, tint_percent);
	if (x + current->get_width() > max_width) {
		current->draw(x - max_width, y, flag, tint_percent);
		if (attach != NULL) {
			attach->x -= max_width;
			attach->show(flag, tint_percent);
			attach->x += max_width;
		}
	}
	else if (x < 0) {
		current->draw(x + max_width, y, flag, tint_percent);
		if (attach != NULL) {
			attach->x += max_width;
			attach->show(flag, tint_percent);
			attach->x -= max_width;
		}
	}
}
```

In this function **show()**, we firstly get the coordinate **x** we got from class **character** to check some conditions with the width and get the width of **current** now by **get_width()** and eventually we will get the coordinate **x** as we want it be and then using **draw()** function from class **Sprite**. The meaning of **flag** here is just used for rotating the object flip horizontal. We then check the boolean **attach** of the doodle and call the virtual function **draw()** from class **Object**. Next we repeat and check other conditions between **x** and **attach** and then using **draw()** and **show()**.

```
void Player::update(float gravity) {
	x += dx;
	y += dy;
	if (attach != NULL) {
		attach->x += dx;
		attach->y += dy;
		if (attach->time_out()) {
			delete attach;
```

```
                        attach = NULL;
                }
                else return;
        }
        if (dy < 0 && dy + gravity >= 0) current = falling;
        dy += gravity;
}
```

This function is used for update the values of the player to check some conditions in game.

***At this game, we consider x,y are the coordinate of the object, dx,dy are the velocity of object(this means postive and negative x will make object go to right and left, positive and negative y will make object go down and up) and the gravity.

Because in this game, doodle always goes down unless the collision or some events happed(the doodle will jump up or rocket) so we also check a condition here to make **dy+=gravity**, this will let the doodle go down

```
int Player::get_width() {
        return current->get_width();
}

int Player::get_height() {
        return current->get_height();
}

float Player::get_y() {
        return y;
}

float Player::get_x() {
        return x;
}

float Player::get_dx() {
        return dx;
}

int Player::get_flag() {
        return flag;
}


float Player::get_dy() {
        return dy;
}
```

Some functions here are just get the values.

```
bool Player::is_falling() {
        return current == falling;
}
```

This boolean is to check whether if doodle is going down or not.

```
void Player::set_y(float _y) {
	if (attach != NULL) attach->y -= (y - _y);
	y = _y;
}

void Player::set_location(float _x, float _y) {
	x = _x, y = _y;
}

void Player::set_dy(float _dy) {
	dy = _dy;
}

void Player::set_dx(float _dx) {
	dx = _dx;
}

void Player::set_flag(int _flag){
	flag = _flag;
}
```

Here are to set the values, different from getting values above.

```
void Player::set_jumping() {
	current = jumping;
}
```

Here is to make the doodle jump, we'll use it for the jumping conditions.

```
bool Player::die(int max_height){
	return y > max_height;
}
```

The game is over if the y-position of doodle pass through the max height we denote in game. This mean the doodle is falling over the display and can not be allowed to check any other conditions and he dies.

Object

```
#include "Sprite.h"

class Player;
class Platform;

class Object
{
protected:
	Sprite *current, *standard, *subordinate;
	Timer *timer;
```

```
	ALLEGRO_SAMPLE *sound;
	float x, y;
public:
	bool trans = false;
	friend Platform;
	friend Player;
	Object();
	~Object();
	int get_width();
	int get_height();
	void set_audio(ALLEGRO_SAMPLE *sound);
	virtual void effect(Player *player) = 0;
	virtual bool allow(Player *player) = 0;
	virtual bool collide(Player *player, int max_width) = 0;
	virtual void show(int flag, float tint_percent) = 0;
	virtual bool time_out() = 0;
};
```

This class is quite little but its virtual function is absolutely important. I don't have to explain the CPP file of this class because it just use for **get_width()** and **get_height()** function, and **set_audio** to add the sound. In this header file, we create 3 **Sprite**s as protected.
Now, these functions **effect(), allow(), collide(), show(),** and **time_out()** we will use them for other classes using functions from **Object**.

Character

```
class Platform;
class Object;

class Character
{
protected:
	float x, y, dx, dy;
public:
	friend Platform;
	friend Object;

	virtual void show(float tint_percent, float max_width) = 0;
	virtual void update(float gravity) = 0;
};
```

This class also has the simple cpp file so we just explain header file. The protected values here are used for the class we explain above **x, y, dx, dy** are absolutely important. In public, we make friend to **Platform** and **Object** to combine to each other easily and these 2 virtual functions **show()** and **update()** we also used in the class above.

Next we will explain 2 effect classes, they are **Rocket** and **Spring**. As being explaining before. The doodle will jump up if it collides to spring and also rocket if collides to rocket. These 2 classes is nearly the same effect is to help doodle jump up without checking any conditions when this effect is still working.
However, in graphics and also a little bit in conditions, the main difference between these 2 classes is class **Spring** is quite simpler because doodle is just tangent and then its jump up with a velocity **dy** and **delay** time but the rocket will still be sticked to the doodle and help it fly up. Now we will explain one by one.

Spring

```
#include "Object.h"
#include "Player.h"

class Spring :
	public Object
{
public:
	Spring(ALLEGRO_BITMAP *img, ALLEGRO_BITMAP *img2);
	void show(int flag, float tint_percent);
	void effect(Player *player);
	bool allow(Player *player);
	bool collide(Player *player, int max_width);
	bool time_out();
	~Spring();
};
```

Here, we include both **Object.h** and **Player.h** in this class. The constructor **Spring** get 2 images, in the main file, we load for this 2 images are the pre-collision spring, and after-collision spring.
We will explain other function in next cpp file.

```
#include "Spring.h"

Spring::Spring(ALLEGRO_BITMAP *img, ALLEGRO_BITMAP *img2) {
	current = standard = new Sprite(img, 1, 1);
	subordinate = new Sprite(img2, 1, 1);
	trans = false;
	timer = NULL;
}
```

Fisrst, we use **Sprite** from inherating from class **Object** to create the 2 images of spring, that is the **current** and **subordinate** from class **Object**. The boolean **trans** and the **timer** are also created in **Object**.

```
void Spring::show(int flag, float tint_percent) {
	current->draw(x, y, flag, tint_percent);
}
```

This function is to draw the spring. The spring will randomly appear on the interface of platforms.

```
bool Spring::allow(Player *player) {
	return player->get_dy() >= 0;
}
```

Boolean function **allow()** to allow the spring to work.

```
void Spring::effect(Player *player) {
	player->set_dy(-30);
	player->set_y(y - player->get_height());
	y -= 3 * 3;
	al_play_sample(sound, 1, 0, 1, ALLEGRO_PLAYMODE_ONCE, NULL);
	current = subordinate;
}
```

In this effect, dy = -30 means that the velocity of player now is -30 and it will jump up(because the direction of y-coordinate is to the bottom) and then **set_y** for the player. Finally, the graphics and sound work, that is the image **subordinate** above change the image of spring. **Y** is set to smaller.

```
bool Spring::collide(Player *player, int max_width) {
	float x2, y2, xp, yp, w, h, wp, hp;
	x2 = player->get_x() + player->get_dx();
	if (player->get_flag() == ALLEGRO_FLIP_HORIZONTAL) x2 += player->get_width() / 3;
	y2 = player->get_y() + player->get_dy();
	w = player->get_width() * 2 / 3;
	h = player->get_height();
	xp = x;
	yp = y;
	wp = get_width();
	hp = get_height();
	bool x_collision = (x2 < xp + wp && x2 + w > xp);
	bool y_pre_collision = player->get_y() + h < yp;
	bool y_collision = y2 + h >= yp;
	if (player->get_x() + player->get_width() > max_width) x_collision = (x_collision
|| x2 - max_width < xp + wp && x2 - max_width + w > xp);
	else if (player->get_x() < 0) x_collision = (x_collision || x2 + max_width < xp +
wp && x2 + max_width + w > xp);
	return x_collision && y_pre_collision && y_collision;
}
```

This function is the most important in this class because of its conditions to allow the effect to realize. In here, **x2, y2** are the x-y coordinate of doodle now including **dx,dy** if it is jumping or falling, **w** and **h** are the size of the doodle image. **Xp, hp** is get from the **x,y** of object and then get the width and height too.
Now, the condition checking are boolean variable. We firstly check the collision of x-coordinate between doodle and spring, then is y-coordinate in both pre and local.

```
bool Spring::time_out() {
	return true;
}
```

Time is out when Doodle finish the jumping progress.

Next is class **Rocket**.


```
#include "Object.h"
#include "Player.h"

class Rocket :
	public Object
{
public:
	friend Player;
	Rocket(ALLEGRO_BITMAP *img, ALLEGRO_BITMAP *img2);
	void show(int flag, float tint_percent);
	void effect(Player *player);
	bool allow(Player *player);
	bool collide(Player *player, int max_width);
	bool time_out();
	~Rocket();
};
```

The header file of class **Rocket** is the same as **Spring** but in here it makes friend with **Player** because of being sticked to doodle. Now we will explain the cpp file.

```
#include "Rocket.h"
#include <iostream>


Rocket::Rocket(ALLEGRO_BITMAP *img, ALLEGRO_BITMAP *img2)
{
	current = standard = new Sprite(img, 1, 1);
	subordinate = new Sprite(img2, 5, 0.2);
	trans = true;
```

```
	timer = NULL;
}


Rocket::~Rocket()
{
}

void Rocket::show(int flag, float tint_percent) {
	current->draw(x, y, flag, tint_percent);
}

void Rocket::effect(Player *player) {
	x = player->get_x() - 7 * 3;
	y = player->get_y();
	player->set_dy(-20);
	if (player->attach != NULL) {
		delete player->attach;
		player->attach = NULL;
	}
	player->attach = this;
	player->current = player->jumping;
	timer = new Timer(2);
	current = subordinate;
}

bool Rocket::allow(Player *player) {
	return true;
}
```

In here, **show()** and **allow()** are the same as the class **Spring** but in the **subordinate** of the rocket, there will have an animation of the rocket.
The effect is also different, first we must get the value **x** and **y** and changing some digit, we also set the velocity is **dy=-20** and now the doodle can not attach to anything, the **current** is also set to **jumping** attitude. The flying time is the **timer** of delay 2 seconds. Finally is changing the graphics of rocket after rocketing.

```
bool Rocket::collide(Player *player, int max_width) {
	float x2, y2, xp, yp, w, h, wp, hp;
	x2 = player->get_x() + player->get_dx();
	if (player->get_flag() == ALLEGRO_FLIP_HORIZONTAL) x2 += player->get_width() / 3;
	y2 = player->get_y() + player->get_dy();
	w = player->get_width() * 2 / 3;
	h = player->get_height();
	xp = x;
	yp = y;
	wp = get_width();
	hp = get_height();
	bool x_collision = (x2 < xp + wp && x2 + w > xp);
	bool y_collision = (y2 < yp + hp && y2 + h >= yp);
	if (player->get_x() + player->get_width() > max_width) x_collision = (x_collision
|| x2 - max_width < xp + wp && x2 - max_width + w > xp);
```

```
	else if (player->get_x() < 0) x_collision = (x_collision || x2 + max_width < xp +
wp && x2 + max_width + w > xp);
	return x_collision && y_collision;
}
```

All the collision conditions is nearly the same as **Spring** as we explain before.

```
bool Rocket::time_out() {
	return (timer == NULL || timer->passed());
}
```

After finish 2 seconds flying, the action is stopped. Now let's go to **Platform**.

Platform

```
#include "Sprite.h"
#include <list>
#include "Player.h"
#include "Spring.h"
#include "Rocket.h"

class Player;
```

To begin, we must mention the class **Player** because we will use this class a lots.

```
class Platform
{
protected:
	Sprite *current, *standard, *subordinate;
	float x, y, dx;
	int flag;
	bool end = false;
	float self_tint = 0;
	Object *attach = NULL;
```

We create some needed variables.

```
public:
	bool avail;
	friend Player;
	int code;
	~Platform();
```

This also makes friend to **Player**.

```
	void show(float tint_percent);

	void update(float offset, float width_limit);
```

Show() and **Update()** functions.

```
	virtual void affect(Player *player, std::list<Platform *> &platforms, int
max_width) = 0;
```

In here, we use **list** to control the platform because it is more useful than **vector** in this situation.

```
	void attach_affect(Player *player);

	void set_flag(int flag);
	void set_location(float x, float y);
	void set_dx(float dx);
	void set_x(float x);
	void set_code(int code);
	void set_attach(Object *obj);

	int get_width();
	int get_height();

	bool out_of_sight(int max_height);
	bool to_be_destroyed();
	bool attach_allow(Player *player);
	bool collide(Player *player, int max_width);
	bool collide_attach(Player *player, int max_width);

	void fix_as(Player *player);
	float get_x();
};
```

For more detail, let's go to explain cpp file.

```
#include "Platform.h"
#include <iostream>

Platform::~Platform()
{
	delete standard;
	if (subordinate != NULL) delete subordinate;
	if (attach != NULL) delete attach;
}
```

We create the destructor first.

```
void Platform::show(float tint_percent) {
	if (attach != NULL) attach->show(flag, tint_percent);
	current->draw(x, y, flag, tint_percent + self_tint);
	if (self_tint < 1) self_tint += self_tint / 10;
	if (self_tint > 1) self_tint = 1;
}
```

At this function, we first check the attach between platform and the object and then **show()** them all. **selt_tint** here is used for some special objects to make the the disappear graphics smoother, for example with the **brown_platform** we will mention later.

```
void Platform::update(float offset, float width_limit) {
	y += offset;
	x += dx;
	if (attach != NULL) {
		attach->y += offset;
		attach->x += dx;
	}
	int w = get_width();
	if (x + w >= width_limit) {
		x = width_limit - w;
		dx *= -1;
	}
	else if (x < 0) {
		x = 0;
		dx *= -1;
	}
}

void Platform::set_flag(int _flag) {
	flag = _flag;
}

void Platform::set_code(int _code) {
	code = _code;
}

void Platform::set_location(float _x, float _y) {
	x = _x;
	y = _y;
}

void Platform::set_dx(float _dx) {
	dx = _dx;
}

void Platform::set_x(float _x) {
	x = _x;
}

int Platform::get_width() {
	return standard->get_width();
}

int Platform::get_height() {
	return standard->get_height();
}

float Platform::get_x() {
	return x;
}
```

These functions are used for updating, setting and getting the value of variables.

```
bool Platform::out_of_sight(int max_height) {
	return y > max_height;
}
```

This boolean is used for checking the platforms which are out of the screen and needless to use again.

```
bool Platform::to_be_destroyed() {
	return end || self_tint == 1;
}
```

Normally with our project, anything related to **tint** almost gets the value of zero but if it is allowed to disappear, the value of tint would be changed to realize.

```
bool Platform::collide(Player *player, int max_width) {
	float x2, y2, xp, yp, w, h, wp, hp;
	x2 = player->x + player->dx;
	if (player->flag == ALLEGRO_FLIP_HORIZONTAL) x2 += player->get_width() / 3;
	y2 = player->y + player->dy;
	w = player->get_width() * 2 / 3;
	h = player->get_height();
	xp = x;
	yp = y;
	wp = get_width();
	hp = get_height();
	bool x_collision = (x2 < xp + wp && x2 + w > xp);
	bool y_pre_collision = player->y + h < yp;
	bool y_collision = y2 + h >= yp;
	if (player->x + player->current->get_width() > max_width) x_collision =
(x_collision || x2 - max_width < xp + wp && x2 - max_width + w > xp);
	else if (player->x < 0) x_collision = (x_collision || x2 + max_width < xp + wp &&
x2 + max_width + w > xp);
	return x_collision && y_pre_collision && y_collision;
}
```

Next, this **collide()** function is also nearly the same as before but different from the figure of height and width according to the image owning them.

```
bool Platform::collide_attach(Player *player, int max_width) {
	if (attach == NULL) return false;
	else return attach->collide(player, max_width);
}
```

This function to check the **attach** and then return to the **collide()** of the **Object**.

```
void Platform::fix_as(Player *player) {
	player->y = y - player->get_height();
}

void Platform::set_attach(Object *obj) {
	obj->x = x + 5 + rand() % int(get_width() - 5 - obj->get_width());
	obj->y = y - obj->get_height();
	attach = obj;
}

void Platform::attach_affect(Player *player) {
```

```
        attach->effect(player);
        if (attach->trans == true) attach = NULL;
}

bool Platform::attach_allow(Player *player) {
        if (attach != NULL) return attach->allow(player);
        else return false;
}
```

These last functions are used for checking attach between **platform** and **object**(here objects can be rocket, spring and the player doodle). With each color of platforms, we have another effect for each. So now turn to the type of **platform**, the main basis of this game.

We have 5 types of platforms: green, brown, blue, white and yellow.


```
#include "Platform.h"

class Green_Platform :
        public Platform
{
public:
        Green_Platform(ALLEGRO_BITMAP *img);
        ~Green_Platform();
        void affect(Player *player, std::list<Platform *> &platforms, int max_width)
override;
};
```

With this **Green_Platform**, this platform is just the basic platform, have no effect and fixed at a postion.

```
#include "Green_Platform.h"

Green_Platform::Green_Platform(ALLEGRO_BITMAP *img)
{
        current = standard = new Sprite(img, 1, 1);
        subordinate = NULL;
        x = y = dx = flag = 0;
        avail = true;
        attach = NULL;
}
```

This uses to load the image of green platform

```
Green_Platform::~Green_Platform()
{
}
```

```
void Green_Platform::affect(Player *player, std::list<Platform *> &platforms, int
max_width) {
	//do_nothing
}
```

Because having nothing to do, we just write do nothing here just for avoiding stress(smile). However, the function must exist to avoid error from using the class **Platform**.

Blue

```
#include "Platform.h"
class Blue_Platform :
	public Platform
{
public:
	Blue_Platform(ALLEGRO_BITMAP *img);
	~Blue_Platform();
	void affect(Player *player, std::list<Platform *> &platforms, int max_width)
override;
	void show(float tint_percent);
};
```

This type of platform is one of 2 funniest platforms. It will move to the left(right) and when it meets the edge, it will go back to the opposite side(here is just changing x-coordinate)

```
#include "Blue_Platform.h"
Blue_Platform::Blue_Platform(ALLEGRO_BITMAP *img)
{
	current = standard = new Sprite(img, 1, 1);
	subordinate = NULL;
	x = y = flag = 0;
	dx = 5;
	avail = true;
	attach = NULL;
}

Blue_Platform::~Blue_Platform()
{
}

void Blue_Platform::affect(Player *player, std::list<Platform *> &platforms, int
max_width) {
	//do nothing
}
```

This platform also does nothing the same as green one.

Yellow

```
#include "Platform.h"

class Yellow_Platform :
	public Platform
{
public:
	Yellow_Platform(ALLEGRO_BITMAP *img);
	~Yellow_Platform();
	void affect(Player *player, std::list<Platform *> &platforms, int max_width);
};
```

```
#include "Yellow_Platform.h"

Yellow_Platform::Yellow_Platform(ALLEGRO_BITMAP *img)
{
	current = standard = new Sprite(img, 1, 1);
	subordinate = NULL;
	x = y = dx = flag = 0;
	avail = true;
	attach = NULL;
}
Yellow_Platform::~Yellow_Platform()
{
}

void Yellow_Platform::affect(Player *player, std::list<Platform *> &platforms, int
max_width) {
	end = true;
}
```

This platform has one effect, that is when doodle jump on it, this platform will disappear.

Brown

```
#include "Platform.h"
class Brown_Platform :
	public Platform
{
public:
	Brown_Platform(ALLEGRO_BITMAP *img, ALLEGRO_BITMAP *img2);
	~Brown_Platform();
	void affect(Player *player, std::list<Platform *> &platforms, int max_width);
};
```

```
#include "Brown_Platform.h"
#include <iostream>
```

```
Brown_Platform::Brown_Platform(ALLEGRO_BITMAP *img, ALLEGRO_BITMAP *img2)
{
	current = standard = new Sprite(img, 1, 1);
	subordinate = new Sprite(img2, 6, 0.01);
	subordinate->allow = false;
	x = y = dx = flag = 0;
	avail = false;
	attach = NULL;
}

Brown_Platform::~Brown_Platform()
{
}

void Brown_Platform::affect(Player *player, std::list<Platform *> &platforms, int
max_width) {
	if (current == standard) x -= (subordinate->get_width() - standard->get_width()) /
2;
	current = subordinate;
	self_tint = 0.1;
}
```

This platform has no meaning, because it happens for nothing. However, it has more commands than other platforms. First, this platform can not be jumped on, if doodle attaches on this, doodle will still fall and the platform disappear with animation and **tint**(to know how tint works, just play the game and see the different, platform does not disappear right away, it needs a little time and fades away).

White

```
#include "Platform.h"
class White_Platform :
	public Platform
{
public:
	White_Platform(ALLEGRO_BITMAP *img);
	~White_Platform();
	void affect(Player *player, std::list<Platform *> &platforms, int max_width)
override;
};


#include "White_Platform.h"
White_Platform::White_Platform(ALLEGRO_BITMAP *img)
{
	current = standard = new Sprite(img, 1, 1);
	subordinate = NULL;
	x = y = dx = flag = 0;
	avail = true;
```

```
}
White_Platform::~White_Platform()
{
}

void White_Platform::affect(Player *player, std::list<Platform *> &platforms, int
max_width) {
	std::list<Platform *>::iterator trace = platforms.begin();
	while (trace != platforms.end()) {
		if ((*trace)->code == code) (*trace)->set_x(rand() % (max_width - (*trace)-
>get_width()));
		trace++;
	}
}
```

Finally is the white platform, this platform is also funny the same as the blue one. When doodle jumps on this, this platform will change its x-coordinate by a small random value(can be considered to “teleport”) and this effect also happens for all the white platforms exist on the display(this means if doodle jumps on one white platform, all other white platforms also “teleport”. We also create for this a special case different from other platforms to make the effect easily occurs.

***That is all of our classes for this project. Now, we turn to explain the **main.cpp** to see how all the classes work and how the game is.

C. Main.cpp

Because this main file is absolutely complicated, we will not mention all the source code to this report and just explain what we think is needed.

```
void init(std::list<Platform *> &platforms, Player *player);
void create_platform(std::list<Platform *> &platforms, bool guarantee = true);
void draw_background();
void draw_addition();
int death_addition();
unsigned to_unsigned(const char * str);
```

We create some needed functions which will be explained later.

```
ALLEGRO_BITMAP *menu_background, *background, *play_on, *score_on, *option_on,
*score_bar, *open_spring, *closed_spring, *rocket_item, *rocket_attached, *pause_icon,
*pause_icon_on, *dead, *death_sign, *flag, *flag_on;
Sprite *flag_sprite, *flag_on_sprite, *current_flag;
ALLEGRO_FONT *font;
ALLEGRO_CONFIG *cfg;
Timer *score_timer;
```

```
ALLEGRO_SAMPLE *sample, *jump, *spring;
```

This step is to create the variables for the images, score, font,…
Now turning to the main program.

```
int main(int args, char **argv) {
	srand(time(NULL));
	al_init();

	al_init_image_addon();
	al_init_primitives_addon();
	al_init_font_addon();
	al_init_ttf_addon();
	al_init_acodec_addon();
	al_install_mouse();
	al_install_keyboard();
	al_install_audio();
	al_reserve_samples(20);
```

First we use the allegro command to addon some init values and install the input variable such as mouse, keyboard,…

```
ALLEGRO_DISPLAY *display = al_create_display(WIDTH, HEIGHT);
	ALLEGRO_EVENT_QUEUE *event_queue = al_create_event_queue();
	ALLEGRO_TIMER *delay = al_create_timer(float(1) / FPS);
	sample = al_load_sample("Audio/Jackpot.wav");
	al_play_sample(sample, 3.0, 0.0, 1.0, ALLEGRO_PLAYMODE_LOOP, NULL);
	jump = al_load_sample("Audio/Jump.wav");
	spring = al_load_sample("Audio/Boing.wav");
```

In here we firstly create the display and then timer,event. After that we load the audio for each effect. In here we loop the sample audio for the background music.

```
	std::list<Platform *> platforms;
	Platform * first_platform = new Green_Platform(imgs[green]);
	first_platform->set_location(FRIST_OFFSET, 500);
	platforms.push_back(first_platform);
```

This step is needed because it makes sure that where the first platform is generated and the player Doodle must jump on it to avoid ending the game from the beginning.

```
Player *player = NULL;
	cfg = al_load_config_file("Init/init.ini");
	player = new Player(cfg, 1);
	my_highscore = to_unsigned(al_get_config_value(cfg, "", "highscore"));

	player->set_location(FRIST_OFFSET + first_platform->get_width() / 2 - player-
>get_width() / 3, 800);
	player->set_dy(-22.5);
```

```
	score_timer = NULL;
```

This step follows the step above to create the player and set its first position. Besides we also create a highscore variable and save it to a ini file which we use **get_config** to get it.

The loop we explain now is very important and it has many source code lines.

```
ALLEGRO_MOUSE_STATE mouse;
                    al_get_mouse_state(&mouse);
                    if (current_state == menu && state == menu) {
                           if (mouse.x >= 89 * 3 && mouse.y >= 44 * 3 && mouse.x <= 128 *
3 && mouse.y <= 59 * 3) {
                                  press_at = game;
                           }
                           else if (mouse.x >= 96 * 3 && mouse.y >= 71 * 3 && mouse.x <=
126 * 3 && mouse.y <= 87 * 3) {
                                  press_at = highscore;
                           }
                    }
```

After create the input for keyboard and mouse, we program the effect of mouse to the menu. Here is to get the position of **Play** and **Highscore** buttons.

```
else if (current_state == game && state == game) {
                           if (mouse.x >= 122 * 3 && mouse.y <= 8 * 3) {
                                  press_at = pause;
                           }
                    }
                    else if (current_state == pause && state == pause) {
                           int dx = al_get_bitmap_width(pause_icon), dy =
al_get_bitmap_height(pause_icon);
                           if (mouse.x >= WIDTH / 2 - dx / 2 && mouse.y >= HEIGHT / 2 -
dy / 2 && mouse.x <= WIDTH / 2 + dx / 2 && mouse.y <= HEIGHT / 2 + dy / 2) {
                                  press_at = game;
                           }
                    }
```

Here is to set the state for pausing game and the position of the icon pause.

```
else if (current_state == game_over && state == game_over) {
                           int add = death_addition();
                           if (mouse.x >= WIDTH / 2 - flag_sprite->get_width() / 2
                                  && mouse.y >= add + death_offset - flag_sprite-
>get_height() + 50
                                  && mouse.x <= WIDTH / 2 + flag_sprite->get_width() / 2
                                  && mouse.y <= add + death_offset - flag_sprite-
>get_height() + 50 + 19 * 3) {
                                  press_at = menu;
                           }
                    }
```

This step just create the graphics effect when the game is over.

```
else if (event.type == ALLEGRO_EVENT_MOUSE_BUTTON_UP) {
                        ALLEGRO_MOUSE_STATE mouse;
                        al_get_mouse_state(&mouse);
                        if (current_state == menu && state == menu) {
                                if (mouse.x >= 89 * 3 && mouse.y >= 44 * 3 && mouse.x <= 128 *
3 && mouse.y <= 59 * 3) {
                                        release_at = game;
                                }
                                else if (mouse.x >= 96 * 3 && mouse.y >= 71 * 3 && mouse.x <=
126 * 3 && mouse.y <= 87 * 3) {
                                        release_at = highscore;
                                }
                        }
                        else if (current_state == game && state == game) {
                                if (mouse.x >= 122 * 3 && mouse.y <= 8 * 3) {
                                        release_at = pause;
                                }
                        }
                        else if (current_state == pause && state == pause) {
                                int dx = al_get_bitmap_width(pause_icon), dy =
al_get_bitmap_height(pause_icon);
                                if (mouse.x >= WIDTH / 2 - dx / 2 && mouse.y >= HEIGHT / 2 -
dy / 2 && mouse.x <= WIDTH / 2 + dx / 2 && mouse.y <= HEIGHT / 2 + dy / 2) {
                                        release_at = game;
                                }
                        }
                        else if (current_state == game_over && state == game_over) {
                                int add = death_addition();
                                if (mouse.x >= WIDTH / 2 - flag_sprite->get_width() / 2
                                        && mouse.y >= add + death_offset - flag_sprite-
>get_height() + 50
                                        && mouse.x <= WIDTH / 2 + flag_sprite->get_width() / 2
                                        && mouse.y <= add + death_offset - flag_sprite-
>get_height() + 50 + 19 * 3) {
                                        release_at = menu;
                                }
                        }
                        else release_at = none;
                }
```

Here is to for what happen when mouse click up after pressing. This is just the corrolary from the key down.

```
else if (current_state == game && press_at == pause) state = pause;
                                else if (current_state == pause && press_at == game) state =
game;
                                else if (current_state == game_over && press_at == menu) {
                                        state = menu;
                                        last_check = 0;
                                        player->set_flag(0);
                                        player->set_location(FRIST_OFFSET + first_platform-
>get_width() / 2 - player->get_width() / 3, 800);
                                        player->set_dy(-22.5);
                                        player->set_dx(0);
                                        platforms.clear();
                                        platforms.push_back(first_platform);
```

```
                        }
```

At this step, we check the conditions for pausing game and resuming and also when the game is over to replay the game.

```
if (current_state == game) {
                                        if (player->get_y() + player->get_height() < LIMIT) {
                                                offset = LIMIT - player->get_y() - player-
>get_height();
                                                player->set_y(LIMIT - player->get_height());
                                                score += offset;
                                        }
```

Now, the score the player has played is added to the variable score which is added from **offset**. In here, offset is get from calculating the distance between the y-postion of player and the limit which player can pass through. This mean if player passes the limit, the display with platform will go down.

```
if (left == right) {
                                                if (player->get_dx() < 0) {
                                                        float new_dx = player->get_dx() +
LEFT_COUNTER_SPEED;
                                                        if (new_dx > 0) new_dx = 0;
                                                        player->set_dx(new_dx);
                                                }
                                                else if (player->get_dx() > 0) {
                                                        float new_dx = player->get_dx() +
RIGHT_COUNTER_SPEED;
                                                        if (new_dx < 0) new_dx = 0;
                                                        player->set_dx(new_dx);
                                                }
                                        }
```

This is to check if the player presses both keys left and right at the same time. So the velocity of x-coordinate **dx** will decrease.

```
else if (left == true) {
                                                player->set_dx(LEFT_SPEED);
                                                player->set_flag(ALLEGRO_FLIP_HORIZONTAL);
                                        }
                                        else if (right == true) {
                                                player->set_dx(RIGHT_SPEED);
                                                player->set_flag(0);

}
```

Now is the moving of doodle, if player presses left or right button, the **dx** will change at the speed we defined at the beginning.

```
if (score >= DISTANCE * difficulty + last_check) {
                                        last_check = score;
                                        create_platform(platforms);
                                        if (DISTANCE * (difficulty +
DIFFICULTY_INCREASE) < MAX_HEIGHT) difficulty += DIFFICULTY_INCREASE;
                                        if (mutate > 2 && rand() % 2 == 0) mutate -= 1;
                                        check = true;
                                }
```

This step is to change the difficulty of the game meaning changing the rate of generating platform.

```
else if (score >= DISTANCE * difficulty / 2 + last_check && score < DISTANCE * difficulty
+ last_check - DISTANCE && check == true && rand() % 2 == 0) {
                                        if (rand() % 2 == 0) {
                                                last_check = score;
                                                create_platform(platforms);
                                                if (DISTANCE * (difficulty +
DIFFICULTY_INCREASE) < MAX_HEIGHT) difficulty += DIFFICULTY_INCREASE;
                                                if (mutate > 2 && rand() % 2 == 0) mutate
-= 1;
                                                check = true;
                                        }
```

However, the difficulty also has a limit which makes sure that Doodle at least has an opportunity to jump on the next platform. But normally, a half of players which played our game always lose from this because of the effect of platforms.

```
if (player->is_falling() && (*plat)->collide(player, HEIGHT)) {
                                        if ((*plat)->avail) {
                                                (*plat)->fix_as(player);
                                                player->set_dy(-BOUNCING_SPEED);
                                                al_play_sample(jump, 0.1, (mouse.x -
WIDTH / 2) / WIDTH, 2, ALLEGRO_PLAYMODE_ONCE, NULL);
                                                need_to_update = false;
                                        }
                                        (*plat)->affect(player, platforms, WIDTH);
                                }
```

In this step, we firstly check whether Doodle is both falling and colliding to platforms or not. If right, Doodle will jump up by **BOUCING_SPEED** and the sound is also released.

```
if ((*plat)->out_of_sight(HEIGHT) || (*plat)->to_be_destroyed() == true) {
                                        plat = platforms.erase(plat);
                                }
```

Now, if the platform is out of the screen, it will be deleted

```
if (current_state == game && player->die(HEIGHT)) {
                                        current_state = state = game_over;
                                        death_offset = HEIGHT;
                                        need_restart = true;
                                }
```

This step is to change the state to gameover when doodle was death.

This is also the end of the loop above and also the main program. Now, let's explain the functions.

```
void init(std::list<Platform *> &platforms, Player *player) {
	Platform * new_platform;
	platforms.clear();
	new_platform = new Green_Platform(imgs[green]);
	new_platform->set_location(rand() % (WIDTH - new_platform->get_width()), HEIGHT -
DISTANCE);
	player->set_location(new_platform->get_x() + new_platform->get_width() / 2 -
player->get_width() / 3, 500);
	new_platform->set_code(green);
	platforms.push_back(new_platform);
```

This function is to generate the platform randomly, in here should be the green one for doodle can jump and has no effect.

```
	Object *obj = NULL;
	for (int h = HEIGHT - 2 * DISTANCE; h >= 0; h -= DISTANCE) {
		if (rand() % 10 == 0) obj = new Spring(closed_spring, open_spring);
		else obj = NULL;
		new_platform = new Green_Platform(imgs[green]);
		new_platform->set_location(rand() % (WIDTH - new_platform->get_width()),
h);
		new_platform->set_code(green);
		if (obj != NULL) new_platform->set_attach(obj);
		platforms.push_back(new_platform);
	}
}
```

We randomly attach the spring to this platforms.

And now is the **draw_background()** function.

```
void draw_background() {
	al_draw_bitmap(background, 0, 0, 0);
	ALLEGRO_MOUSE_STATE mouse;
	al_get_mouse_state(&mouse);
```

We draw the background for game and also add the state of mouse.

```
	if (current_state == menu) {
		al_draw_tinted_bitmap(menu_background, al_map_rgba_f(1, 1, 1, 1 -
tint_percent), 0, 0, 0);
```

The step above is to load the menu_background to play or see the highscore we explained before.

```
		if (mouse.x >= 89 * 3 && mouse.y >= 44 * 3 && mouse.x <= 128 * 3 && mouse.y
<= 59 * 3) al_draw_tinted_bitmap(play_on, al_map_rgba_f(1, 1, 1, 1 - tint_percent), 89 *
3, 44 * 3, 0);
		if (mouse.x >= 96 * 3 && mouse.y >= 71 * 3 && mouse.x <= 126 * 3 && mouse.y
<= 87 * 3) al_draw_tinted_bitmap(score_on, al_map_rgba_f(1, 1, 1, 1 - tint_percent), 96 *
3, 71 * 3, 0);
		if (mouse.x >= 89 * 3 && mouse.y >= 166 * 3 && mouse.x <= 124 * 3 &&
mouse.y <= 181 * 3) al_draw_tinted_bitmap(option_on, al_map_rgba_f(1, 1, 1, 1 -
tint_percent), 89 * 3, 166 * 3, 0);
```

These steps are just use **tint** effect for impressive view.

```
		if (score_timer != NULL) {
			al_draw_text(font, al_map_rgb(score_tint, score_tint, score_tint),
107 * 3, 116 * 3, 0, std::to_string(my_highscore).c_str());
			if (inc == true) {
				if (score_tint + 2 <= 255) score_tint += 2;
				else inc = false;
			}
			if (inc == false) {
				if (score_tint - 2 >= 0) score_tint -= 2;
				else {
					inc = true;
					delete score_timer;
					score_timer = NULL;
				}
			}
		}
	}
```

This step is to show the score on the display when playing game.

```
	else if (current_state == pause) {
		int dx = al_get_bitmap_width(pause_icon), dy =
al_get_bitmap_height(pause_icon);
		if (	mouse.x >= WIDTH / 2 - dx / 2
			&& mouse.y >= HEIGHT / 2 - dy / 2
			&& mouse.x <= WIDTH / 2 + dx / 2
			&& mouse.y <= HEIGHT / 2 + dy / 2)
al_draw_tinted_bitmap(pause_icon_on, al_map_rgba_f(1, 1, 1, 1 - tint_percent), WIDTH / 2
- dx / 2, HEIGHT / 2 - dy / 2, 0);
		else al_draw_tinted_bitmap(pause_icon, al_map_rgba_f(1, 1, 1, 1 -
tint_percent), WIDTH / 2 - dx / 2, HEIGHT / 2 - dy / 2, 0);
	}
```

In this step, the pause button on the display works and the game temporarily stops.

```
	else if (current_state == game_over) {
		int add = death_addition();
		if (mouse.x >= WIDTH / 2 - flag_sprite->get_width() / 2
```

```
                    && mouse.y >= add + death_offset - flag_sprite->get_height() + 50
                    && mouse.x <= WIDTH / 2 + flag_sprite->get_width() / 2
                    && mouse.y <= add + death_offset - flag_sprite->get_height() + 50 +
19 * 3) {
                    if (current_flag == NULL) current_flag = flag_on_sprite;
                    else if (current_flag != flag_on_sprite) flag_on_sprite-
>get_current_of(current_flag);
                    current_flag = flag_on_sprite;
             }
             else {
                    if (current_flag == NULL) current_flag = flag_sprite;
                    else if (current_flag != flag_sprite) flag_sprite-
>get_current_of(current_flag);
                    current_flag = flag_sprite;
             }
             current_flag->draw(WIDTH / 2 - flag_sprite->get_width() / 2, add +
death_offset - flag_sprite->get_height() + 50, 0, tint_percent);
             al_draw_tinted_bitmap(death_sign, al_map_rgba_f(1, 1, 1, 1 - tint_percent),
WIDTH / 2 - al_get_bitmap_width(death_sign) / 2, 100, 0);
             if (death_offset >= HEIGHT - al_get_bitmap_height(dead) + 40) death_offset
-= 0.1;
             al_draw_tinted_bitmap(dead, al_map_rgba_f(1, 1, 1, 1 - tint_percent), WIDTH
/ 2 - al_get_bitmap_width(dead) / 2, add + death_offset + 10, 0);
```

These next steps are used for draw the screen when the game is over. Play the game and lose to see how it looks like.

```
             if (int(score) > my_highscore) {
                    my_highscore = int(score);
                    al_set_config_value(cfg, "", "highscore",
std::to_string(my_highscore).c_str());
                    al_save_config_file("Init/init.ini", cfg);
             }
             al_draw_text(font, al_map_rgb(37, 34, 99), WIDTH / 2 -
al_get_bitmap_width(death_sign) / 2 + 55 * 3, 100 + 9 * 3, 0,
std::to_string(my_highscore).c_str());
             al_draw_text(font, al_map_rgb(37, 34, 99), WIDTH / 2 -
al_get_bitmap_width(death_sign) / 2 + 58 * 3, 100 + 18 * 3, 0,
std::to_string(int(score)).c_str());
       }
}
```

We have created the highscore variable before and saved it to ini file. Now the player has archied a new score, we firstly read the old highscore from the ini file (we did at the beginning) and then compare to the score now, if the score is greater than highscore, it becomes the new high score and then we save it.

```
void draw_addition() {
       if (current_state == game) {
             al_draw_tinted_bitmap(score_bar, al_map_rgba_f(1, 1, 1, 0.7), 0, 0, 0);
             al_draw_text(font, al_map_rgb(255, 255, 255), 4, 3, 0,
std::to_string(unsigned(score)).c_str() );
```

```
	}
}
```

This is to draw the score bar at the top of the screen.

```
int death_addition() {
	ALLEGRO_MOUSE_STATE mouse;
	al_get_mouse_state(&mouse);
	if (mouse.y <= HEIGHT / 2) return 0;
	return - (mouse.y - HEIGHT / 2) * 45 / (HEIGHT / 2);
}
```

This is also additional for the effect when the game is over like this.

Figure 3. The menu of Doodle Jump

Player moves the mouse pointer to the bottom, the image appears(amazing).

```
unsigned to_unsigned(const char * str) {
	unsigned ans = 0;
	for (int idx = 0; str[idx] != '\0'; idx++) ans = ans * 10 + (str[idx] - '0');
	return ans;
}
```

This function is just like the convert function we did in the class Player. This is just to convert from config file to get the number we need.

Now is most important function, to create the platform.

```
void create_platform(std::list<Platform *> &platforms, bool guarantee) {
	Platform * new_platform;
	Object *obj = NULL;
	static bool got_white = false;
	static int cool_down = 5;
```

```
	int pick;
	if (got_white == true) {
		cool_down -= 1;
		if (cool_down > 0) {
			pick = white;
			goto skip;
		}
		else {
			got_white = false;
			cool_down = rand() % 5;
		}
	}
```

This is special case for white platform as we explained at class White_Platform. If the white platform is generated, there are other white platform are generated too to make the effect: All white platform which can be seen on the display will “teleport”.

```
	if (guarantee == true) {
		if (rand() % mutate == 0) {
			if (mutate > 2 && rand() % 2 == 0) mutate -= 1;
			pick = Guarantee[rand() % (sizeof(Guarantee) /
sizeof(Guarantee[0]))];
		}
		else pick = green;
	}
	else {
		if (rand() % 1 == 0) pick = brown;
		else if (rand() % 3 == 0) pick = yellow;
		else if (rand() % mutate == 0) {
			if (mutate > 2 && rand() % 2 == 0) mutate -= 1;
			pick = rand() % END;
		}
		else pick = green;
	}
	if (pick == white) got_white = true;
```

This step is to make sure that, when the game get harder, Doodle always has at least one opportunity to jump on.

```
skip:
	switch (pick) {
	case green:
		new_platform = new Green_Platform(imgs[green]);
		if (rand() % 10 == 0) {
			obj = new Spring(closed_spring, open_spring);
			obj->set_audio(spring);
		}
		else if (rand() % 20 == 0) obj = new Rocket(rocket_item, rocket_attached);
		break;
	case blue:
		new_platform = new Blue_Platform(imgs[blue]);
		if (rand() % 5 == 0) {
```

```
                        obj = new Spring(closed_spring, open_spring);
                        obj->set_audio(spring);
                }
                else if (rand() % 30 == 0) obj = new Rocket(rocket_item, rocket_attached);
                break;
        case white:
                new_platform = new White_Platform(imgs[white]);
                break;
        case brown:
                new_platform = new Brown_Platform(imgs[brown], imgs[dis_brown]);
                break;
        case yellow:
                new_platform = new Yellow_Platform(imgs[yellow]);
                break;
        default:
                new_platform = new Green_Platform(imgs[green]);
                break;
        }
        if (pick == blue) {
                new_platform->set_dx(5);
        }
        new_platform->set_location(rand() % (WIDTH - new_platform->get_width()), -
new_platform->get_height());
        new_platform->set_code(pick);
        if (obj != NULL) new_platform->set_attach(obj);
        platforms.push_back(new_platform);
}
```

This step we just separate cases for each type of platform, we also add the spring and rocket to this.

***This is also the last part of Code Explanation. However, please note that we did not do this explanation after finishing 100% of the game. This means we also adjusts and modified some variables, functions and also added more things, such as the sound. In overall, the project is working well but maybe we lost something and a little bit difference between the explaination in report and the source code could appear but it does not matter probably.

6. Conclusion

In conclusion, we wish that our project will make you impress. Through the developing time of this game we had some problems but luckily found out the way to solve. We have checked the game many times before sending it to you and hope that it can perform well when you test it. Finally, we hope our skills and knowledge in this programming language will follow us to the entire life to help us in career latterly, not only in this project.

We appreciate the time you have taken to review this report and we hope you will feel free to contact us if you have any questions at the E-mail:

EEIT2016_Viet.NPB@student.vgu.edu.vn **EEIT2016_Luan.NTM@student.vgu.edu.vn**

Sincerely yours,

Nguyen Phan Bao Viet Nguyen Tran Minh Luan

7. References

Firstly, we have to thank our friend Thang Le Huu for helping us in this game so much, from recommendaion of the Allegro library and command to help us in some algorithms and the graphics for images.

Secondly, the lectures from Dr.Nga is very useful for us to learn OOP in C++

Here is other references:

https://www.allegro.cc/

https://www.visualstudio.com/

http://www.learncpp.com/