



Vietnamese-German University

High Level Programming course

C++ Project Final Report

Ultimate Chicken War

An Object-oriented software, turn-based game with entertaining functionalities.

To: Dr. Nga Tran Phuong

Team members: Nguyen Trong Khiem (11469) & Vo Hong Phuc (12119)

Major: EEIT2016

Date: 01/06/2018

Abstract

In our high-level programming language course provided by VGU, we are requested to build a software that implements C++ and its *object-oriented programming* functionalities. Therefore, we decided to design a game that allow us to practice the aforementioned theory we learnt in class and get ourselves familiar with many additional graphic library like *SDL2*.

The following is in response to our instructor's requirement in 26 February 2018 that we write a final report describing our C++ project.

Keyword: Object-oriented programming, game, turn-based, pixel, Piskel, classes, inheritance, polymorphism, C++, SDL2, Visual Studio, Github

Table of contents

- I. Overview
- II. Game instruction and functionalities
- III. Class hierarchy
- IV. Designing concept
- V. Code detail explanation
 - A. UCW - Game system controller
 - 1. Main.cpp
 - 2. UCW classes
 - B. Player system
 - 1. Player class
 - 2. Chicken class
 - 3. Component
 - 3.1 Component class
 - 3.2 SpriteComponent : public Component
 - 3.3 TransformComponent : public Component
 - 3.4 StatsComponent : public Component
 - C. UI - User interface
 - 1. Menu : public UI
 - 2. NameInput : public UI
 - 3. Popup : public UI
 - 4. Stats_table : public UI
 - 5. Map : public UI
 - D. Turn-based algorithm - Mouse controller
 - E. Additional classes for other purposes
 - 1. TextureManager
 - 2. LoadTextureFromText
 - 3. Vector2D
- VI. Self evaluation

I. Overview

Ultimate Chicken War (UCW) is a 2D turn-based game whose A **turn-based** strategy (TBS) **game** is a strategy **game** (usually some type of wargame, especially a strategic-level war-game) where players take turns when playing. Our game requires 2 players to navigate their characters through our self-designed map and fight with other players.

The platform that we used for coding and exchanging codes are Microsoft Visual Studio 2015 and Github. To aid the visual effect like rendering images and window, we make use of the SDL2 library. To design the map, characters and items, we use the pixel - painting software, PISKEL. The image is then exported into PNG format so than the transparency can be remained.







In the process of coding, we had to counsel 2 main sources of tutorial, the first one is <http://lazyfoo.net/>, which guided us on how use SDL2 library, and the other is Let's Makes Games video series on Youtube on how to design a *Entity Component System (ECS)* in C++: https://www.youtube.com/watch?v=QQzAHcojEKg&list=PLhfAbcv9cehhkG7ZQK0nfIGJC_C-wSLrx

If you want, you can visit out Github repo at <https://github.com/cruzion/UltimateChickenWar>

There is an application file .exe for you to run. However, to compile our code, please install *Windows 10 version 10.0.10240.0* or higher, *Visual Studio 2015 (v140)*, *Debug Mode x86*.

II. Game instruction and functionalities

Since our games is turn-based, there should be 2 *players* for this game. There are 5 maps to be randomed. The 3 *white* chickens on the *top left corner* will represent the *player 1*. The 3 *black chickens* on the *bottom right corner* will represent will represent *player 2*. Each of the 3 chickens from each player will have different abilities, avatar and stats, which are described by this table:

Chicken	Health point (hp)	Attack (atk)	Defend (def)	Range
 & 	3	2	1	1
 & 	3	1	0	2
 & 	3	1	2	1

If one chicken attacks another, the others' stats is calculated as follow:

- $Opponent's\ HP = Opponent's\ HP + Opponent's\ DEF - Your\ ATK$ (if its DEF is smaller than your ATK)
- $Opponent's\ DEF = Opponent's\ DEF - Your\ ATK$ (if its DEF is greater than your ATK)

Next, each player will *take turn* to move 1 of their chickens, starting with player 1.

The *red hexagon* surrounding the tile will indicate *which* chicken is allowed to move, which is also indicated as the stats line turn *red* in the table.

As the player moves the mouse across the map, a *blue hexagon* will show that the chicken is allowed to move or attack there, which is dependent on the chicken's range. Click to that tile to move the chicken there.

If nothing is shown, then the chicken **cannot** move there.

If there is another chicken standing there (including other chickens in your team too!), you can **attack** it or not by clicking on that chicken. There will be a **popup** table in the middle underneath the map, on which you can click "attack" to attack that chicken or click on "not" to not do so, which doesn't count as 1 turn.

Only attacking and moving will be counted as 1 turn.

Also, there are items on the map for you to collect and enhance your chickens' stats. Click on that tile to move your chicken there and collect it.

You can click on the chicken itself to skip its turn.

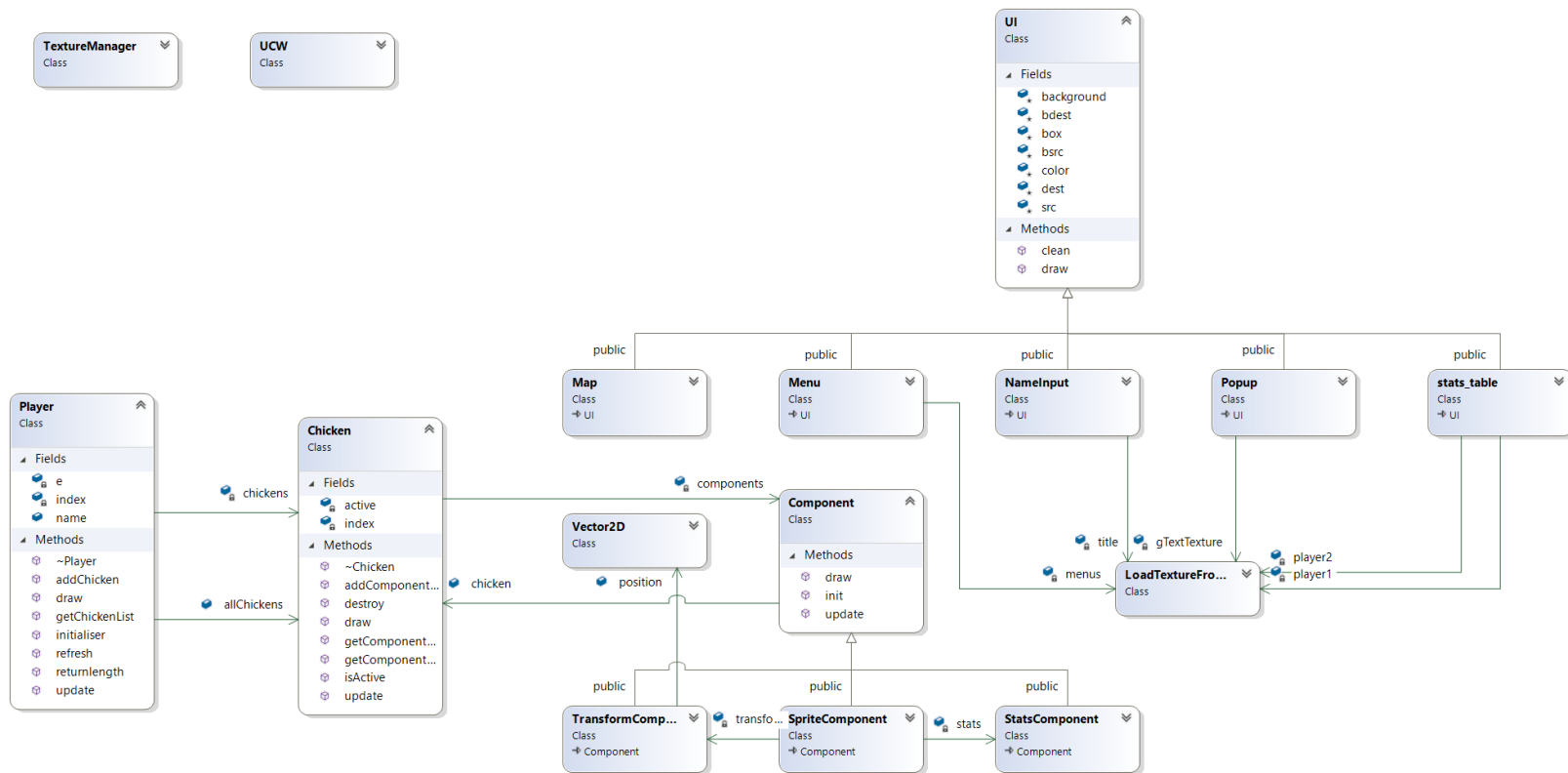
There are water tiles on the map and your chicken **cannot** move there.

The player that no longer has any chicken on the table is the **loser**.

For more details, please have a look at our *readme.md* file we included with this report.

III. Class hierarchy

Our game has 5 main base classes: *UCW* for game management, *Player* for player management, *Chicken* for chicken management, *Component* for component management of each chicken and *UI* for controlling User Interface components. Each has its own derived class. There are also some other classes used for other functions like *TextureManager*....:



Most of the classes have a “*Has-a*” relationship with *SDL2* classes. However, for the sake of simplicity, we will not mention it here since it is very obvious in the code and unreasonably lengthens the report.

In the diagram, the arrows with white heads demonstrate “*is-a*” relationship; arrows with variable name on top of it and black heads demonstrate “*knows-a*” relationship (“*knows-a*” relationship means an object of this class contains the *pointer* or *reference* to another object of another class). Since there are so many “*has-a*” relationships, we couldn’t display it here. They are represented by the Fields and Methods compartment of the box of each class (Like *Player* and *Chicken* on the diagram here).

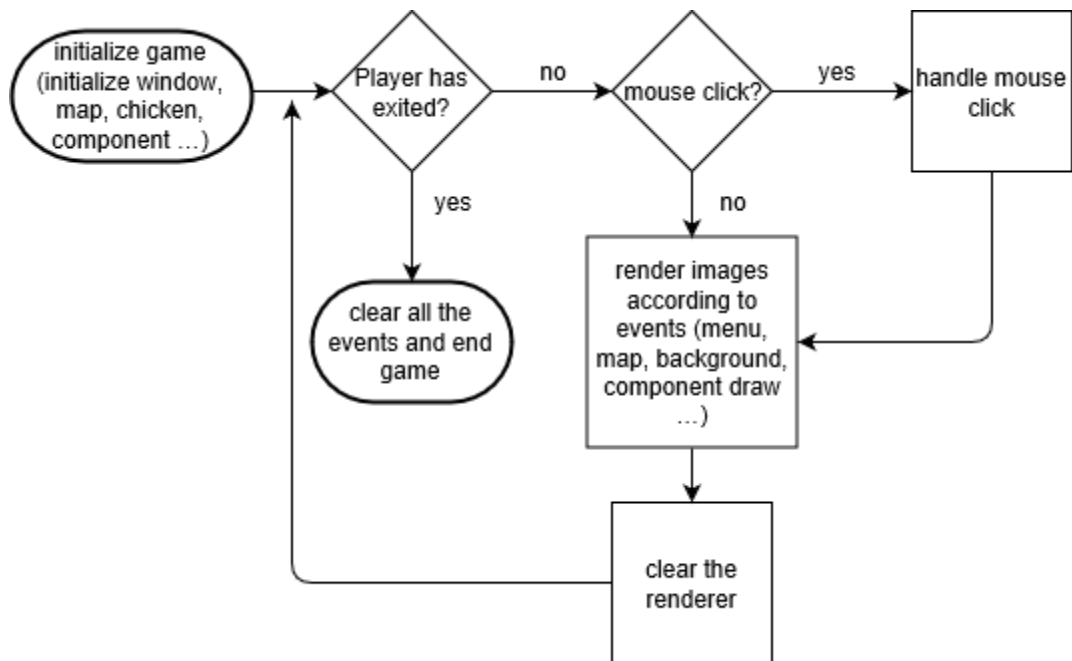
If you want to have a more detailed view about our class hierarchy, please open the *ClassDiagram.cd* file in our *Project Solution* on *Visual Studio*.

IV. Designing concept

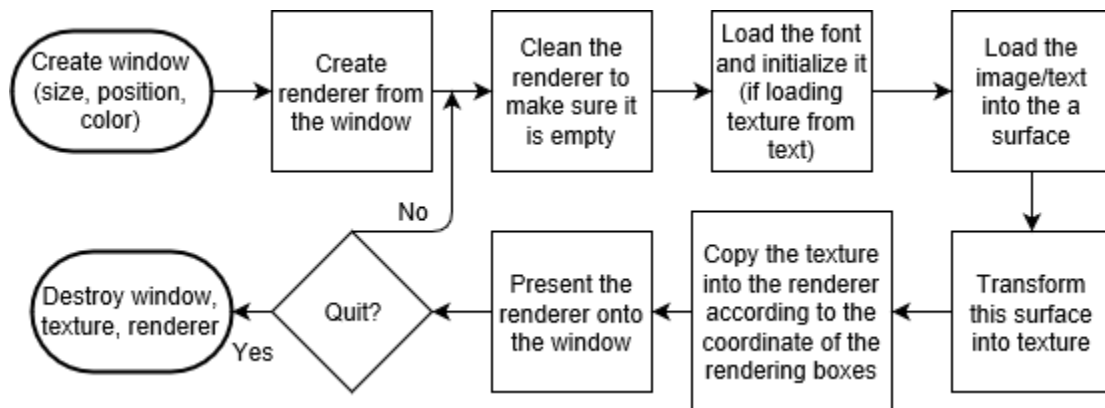
Here is the logic of our game:

- We designed map tiles in *hexagon blocks* so that within a same area, a player can have more directions to move (6 vs 4)
- We set out the *UCW* as the platform for our gameplay (like a *main()* function, except that *UCW* has methods and member fields), meaning this class will be an independent class which control all the events, classes, player, UI of the game.
- A game must have players, so we set out the *Player* class doing this job.
- Since there are chickens as main characters in the game, we have to make a *Chicken* class.
- *Player* has chickens, so chickens must be in the member fields of players. Both players can share each other about their chickens, so we made a static variable which store infos about all 6 chickens and also outsiders can access them.
- Chickens, like humans, must have characteristic, so we make the *Component* class as the member of the *Chicken* class. A component **cannot** be possessed by 2 chickens (e.g. different chickens has different images), therefore we need each component to points to only **1** master, so we created a pointer to point to that chicken as a member of a component.
- The chicken must have an image (*SpriteComponent*), and some ability like moving ability (*TransformComponent*), attacking ability, defending ability (*StatsComponent*), so we create some derived classes of the *Component* class.
- Since the game must also have an *User Interface*, we create a mother class called *UI*. Since we need the *UI* to be displayed on the screen, it need to have a *rendering box* and the *color* so we made them as the common members of all the UIs.
- Since a UI is something that the players can see and interact, this means the menu, the table displaying the chicken's stats, the Map, ... are all UIs.
- Sometimes, we need to display some important messages during the game so we need to have a *Popup* class.
- Most of the activities in the games are done by mouse, so we need to have a *Mouse_Controller* file doing this job.
- Players need to know **when** is their **turn**, and also **where** they can move, so we add some *indicator hexagons* to show this
- Players need to know the stats of their chickens, so we display a stats table.
- There should be some items as well as difficult terrains on the map so that the game can be more challenging, involving more strategy, risk and rewards.
- Different random maps to make game more vivid and interesting.

Game flow:



SDL2 mechanism



V. Code detail explanation

We will explain some important parts of the code. Some trivial parts, variables, functions will be overlooked in here and explained in the comment of the code

A. UCW - Game system controller

- Purpose: Control the overall events in the game

1. *Main.cpp*

- We define some information of the game like FPS (frame per seconds), and *frameDelay* in case the the frame runs too fast.

```
const int FPS = 60;
const int frameDelay = 1000 / FPS;
```

- Then we point the *UCW *game* pointer to a dynamic memory to create an instance of the *UCW* class. Then we initialize the game window at the center of the desktop, with a name and a fixed size but we an also resize it by using resize button on window or use mouse to change the scale of the window without affecting on the game.

```
game = new UCW(); //create a game
game->init("ULTIMATE CHICKEN WAR", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 1152, 600); //init the game
```

- Then, we run into a while loop as long as the condition that the game has not ended is met. We will respectively start counting the time of the frame, handle every events in the game instance, update them and render all of them onto the window. Also, if the time of the frame is too long, we will delay it:

```

while (game->running()) {
    frameStart = SDL_GetTicks();

    game->handleEvents();
    game->update();
    game->render();

    frameTime = SDL_GetTicks() - frameStart;

    if (frameDelay > frameTime) {
        SDL_Delay(frameDelay - frameTime);
    }
}

```

- If the game has ended, we will check if you pressed restart. If yes, then goto the beginning of the code, otherwise we just delete the game pointer dynamic memory and quit the game.

```

bool restart = game->restart;
game->clean();

delete game;
if (restart == true) goto reset;

```

2. UCW classes

- UCW represents the game controller unit.
- Some important variables:

```

Player manager1;
Player manager2;
Chicken* player1[3];
Chicken* player2[3];

```

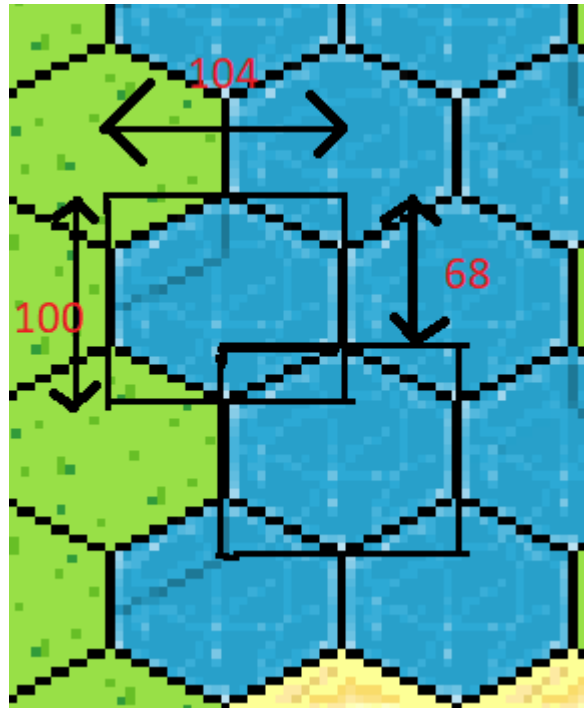
- Manager1, manager2 represents the two players (UCW.cpp).
- Player1, player2 represents the chicken list of each player (UCW.cpp).
- Also, we use some static member like the renderer and event that every instance of UCW is going to share and outside function can access them to:

```
static TTF_Font *gFont;  
static SDL_Renderer* renderer;  
static SDL_Event event;
```

- UCW::init():
 - First, we initialise the window as specified by the argument lists.
 - Then we have to ensure that our game can run on any resolution:

```
SDL_SetHint(SDL_HINT_RENDER_SCALE_QUALITY, 0);  
SDL_RenderSetLogicalSize(renderer, 1728, 900);
```

- Then, we want to make sure the chickens (*Chicken* player1[3]*, *Chicken* player2[3]*) are not pointing to anything by calling the `destroy()` method of the *Chicken* classes. Afterward, we initialise the public static variable *allChickens[6]*, which contains all chickens of both sides by calling the static method `initializers()`.
- Then we create an instance of the *Map* class called *GameMap* for loading the map. Also, we set the `first_time` variable to be true, indicating that the next mouse click is the first time.
- Also, we calculate the coordinates (row & columns) of the bounding box that we are going to render each tile



- We then pass the pointer to each chicken to the player1 and player2 array by calling the *addChicken()* method of the Player class, which return the pointer to the newly created Chicken instance:

```
player1[i] = manager1.addChicken(2*i);
```

- Afterwards, we add the component to each chicken by passing the typename into the *addComponent* template function:

```
player1[i]->addComponent<TransformComponent>(position_ini[2*i], 1);
player1[i]->addComponent<SpriteComponent>(image[2*i], turn_indicator, range_indicator);
player1[i]->addComponent<StatsComponent>(2*i, stats_array[i][0], stats_array[i][1], stats_array[i][2], stats_array[i][3]);
```

- Also, we set the first turn to move to be the first chicken from player1:

```
player1[0]->getComponent<StatsComponent>().myturn = true;
```

- *Void UCW::handleEvents():*
 - We use *SDL_PollEvent(&event)* to wait for any event happen in the game (like keypress or mouse motion)
 - If the event is Quit, then we quit the game
 - If the event is a mouse click, then we will run the move the chicken accordingly in the

Mouse_Controller function.

- *Void UCW::update()*: This function will iteratively refresh update both players information
- *Void UCW::render()*: This functions will load and render the requested images from the
 - First we load the background from an PNG image using the static function *TextureManager::LoadTexture*.
 - Then we clear the renderer before we draw the background.
 - Then it we check if you have pressed *Start* or *Exit* in the menu, if yes then we draw the *nameInput* popup. Also, if you press restart, then we will set *int UCW::restart* to be true and *int UCW::isRunning* to be false.
 - After both players have input their names, we then draw the map by calling the *draw()* method of both players. Also, we will draw the stats table describing the status of each player.
 - Afterwards, we will check whether a chicken is attacking others, meaning we will have display the popup showing “Attack or not”:

```
for (int i = 0; i < 6; i++) {  
    if ((Player::allChickens[i])->getComponent<StatsComponent>().choosing) {  
        draw_allowing = true;  
        break;  
    }  
}
```

- Also, we need to count the number of chickens left on the map of chicken on each side:

```
for (int i = 0; i < manager1.returnlength(); i++) {  
    if (player1[i]->getComponent<StatsComponent>().isAlive) numPlayer1++;  
    if (player2[i]->getComponent<StatsComponent>().isAlive) numPlayer2++;  
}
```

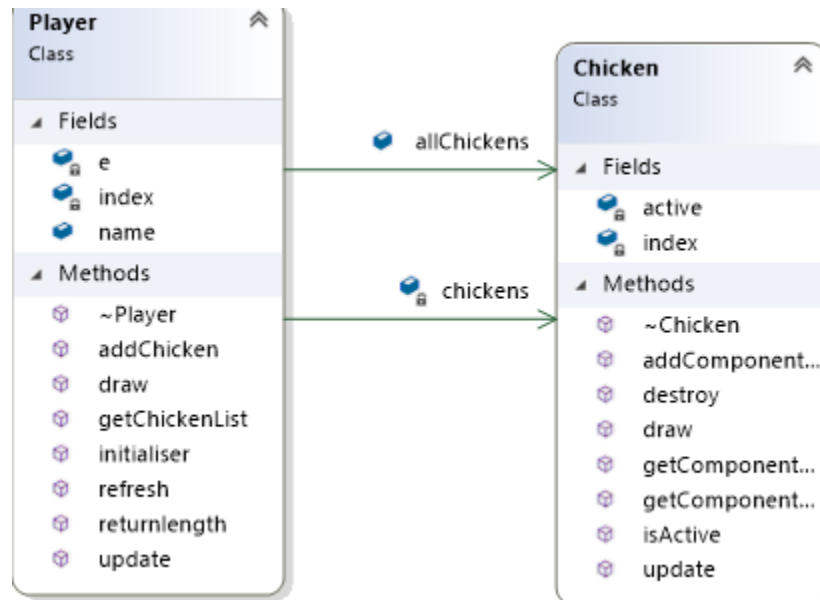
- Finally we have to clean everything we have rendered to make sure the next time *UCW::render()* is called, the renderer is empty.
- *Void UCW::clean()*: This function clean all process and quit the game

B. Player system

- Purpose: Control what each player can do

1. Player Class

- Representing the player.
- Relationship with other classes:

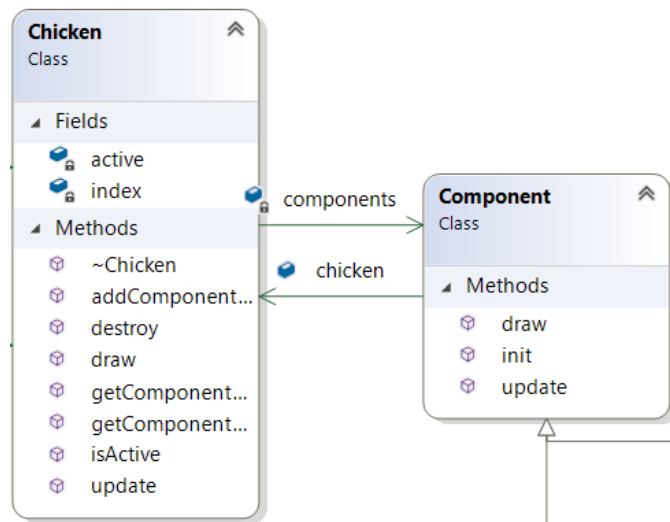


- `Player::**chickens` represent the pointers to 3 chickens of each player.
- `Static Player::*allChickens[6]` is a list of all 6 chickens that 2 player shares.
- static void `initialiser()` is used to initialize the static `allChickens` pointer array to `NULL`. Note that static function can only access static variable and can be called without using any instance like `Player::initialiser()`;
- Has a friend relationship with `UCW::init()` since this will access private members of Player class.
- `Player::~~Player()`: This destructor is used to release all the dynamic memory.
- `Player::update()` is used to update every chicken's component and `Player::refresh()` is used to check which chicken is alive.
- `Chicken* addChicken(int i)`:
 - First we create a pointer `Chicken::e` to a new instance of Chicken
 - Since we need to increase the size of an array, we create a new temporary array of pointer with size specified by `index` (larger than old array by 1). By doing this, we are not limited with the number chicken of each sides since we can increase

- the size of the array freely.
 - Then we swap the old one with new one, after which we assign the e pointer to the last index of the array.
 - We delete the temporary array and return the pointer e
- Player::Chicken** getChickenList()*: return the chicken list of each player
- int Player::returnlength()*: return the length of the chicken list by checking if the pointer is null or not

2. Chicken class

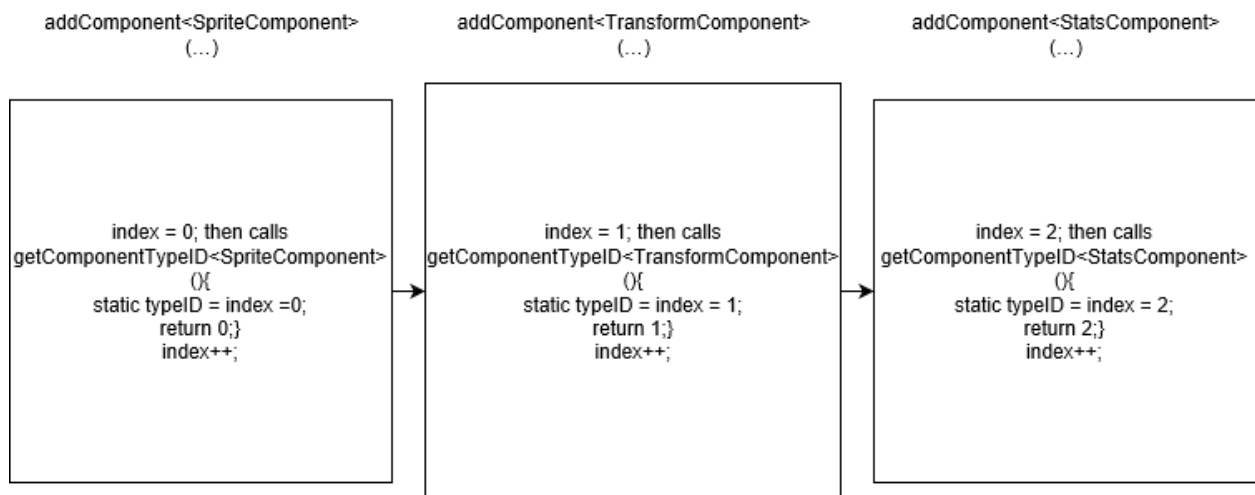
- Represent the chicken
- Relationship with other classes:



- Chicken::Components[3]* represent the 3 components of chicken
- Chicken::~~Chicken()*: destroy the dynamic memories that we used in filin
- void Chicken::update()* : Update each component of the chicken
- Void Chicken::draw()*: Draw each component of the chicken
- Bool Chicken::isActive()*: Check if the chicken is active
- Bool Chicken::destroy()*: Destroy the chicken
- Template <typename T, typename... Targs> void Chicken::addComponent()*: Each time we need to add a component, we pass in the name of that component (T), and all the parameter we need to initialize the constructor of that component class (we use ...mArgs to indicate unlimited parameters):
 - We then create a pointer c point to the newly created component instance (we need to expand the arguments into the constructor by writing mArgs....)
 - We also assign the chicken that possess that component is “this” (“this” is the chicken

that is calling this method)

- Then, we add this newly created component pointer to the component array. We have to use `static_cast` since this is an upcast (T is derived class of the base class `Component`) and we know this is a validcast so we don't need the compiler to check it.
- We also need to give the id to this component by calling the template function `GetComponentTypeID()`. Each time pass in the value T of the template (e.g. `TransformComponent`), a brand new function `GetComponent` is created (e.g. `TransformComponent& GetComponentTypeID()`). If this is the first time this template of this type T is called, the `TypeID` of that component will be initialised with the index (static variable is only called once at the beginning, and will store the same value unless it is changed intentionally). Therefore, this component's id will remain the next time this template function of the same type is called. Also, we need to increment the index too so that different component will have different id.
- After provide the component with an id, we need to initialise the component by calling the method `init()` of the component class.



- `template<typename T> T* Chicken::GetComponent():` Return the pointer of the T-type component:
 - First, we need to get that component out of the component array by calling the `GetComponentTypeID<T>()` to get the index of the component in the array.
 - Then we need to `dynamic_cast` the downcast pointer back to the derived class T since the pointer we get from the array is of `Component*` type, which is the base class T*.

getComponent<SpriteComponent>
(...)

getComponent<TransformComponent>
(...)

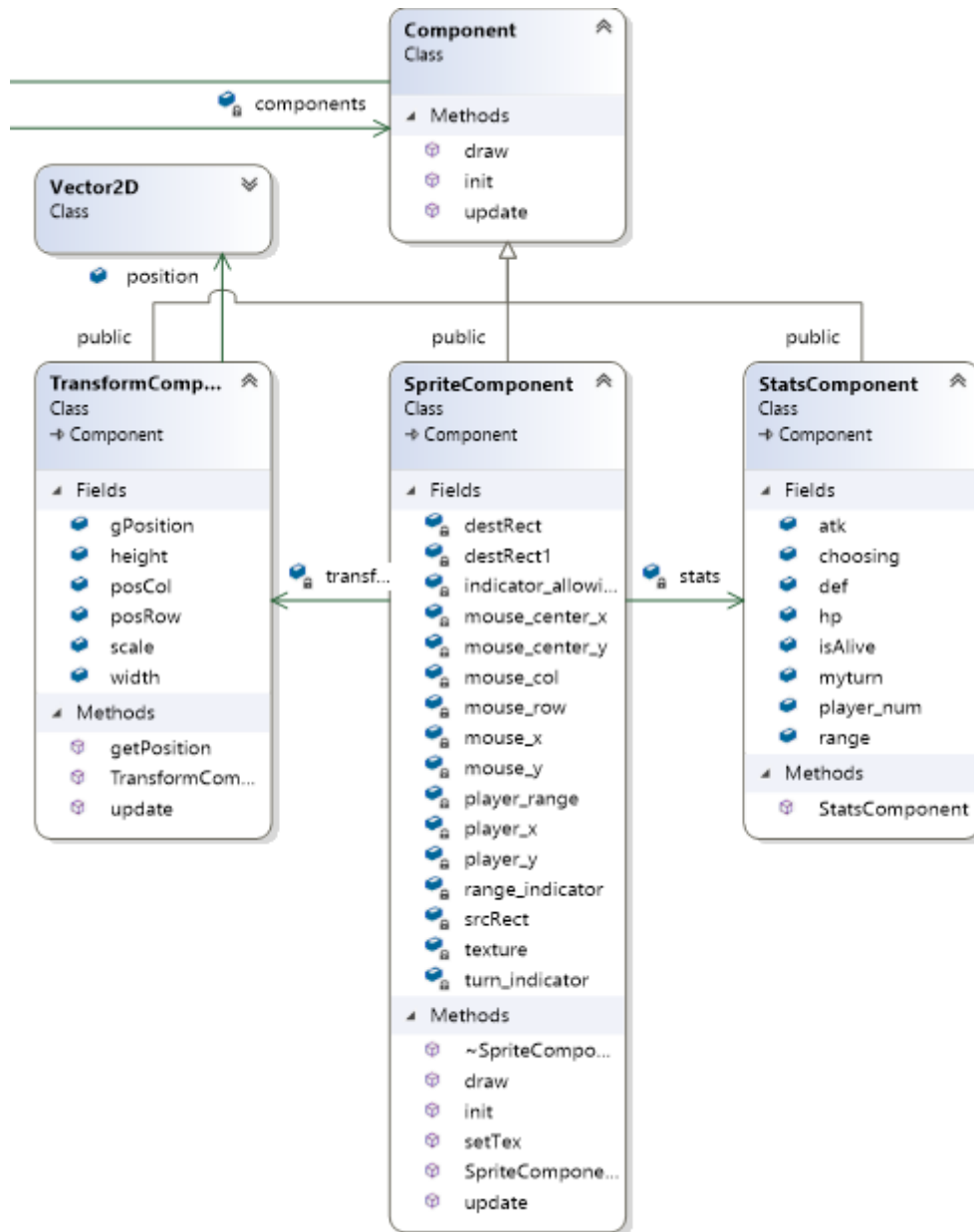
getComponent<StatsComponent>
(...)

calls to
getComponentTypeID<SpriteComponent>
{
 typeID = 0;
 return 0;}

calls to
getComponentTypeID<TransformComponent>
{
 typeID = 1;
 return 1;}

calls to
getComponentTypeID<StatsComponent>
{
 typeID = 2;
 return 2;}

3. Component classes



3.1 Component base class

- Control all the component of each chicken. Make use of polymorphism so that the code can be more systematic and simple.
- Has “Knows-a” two-way relationship with *Chicken* class (*Chicken* has **Component* but *Component* has *Chicken** that points to its owner). This allows each component access each other easily.
- *Component::Chicken** chicken: The chicken that has this component.
- *virtual Component::init()*, *virtual Component::update()*, *virtual Component::draw()*: this are function to be overloaded in the derived class. For example, if we call *Chicken :: components[1]* -

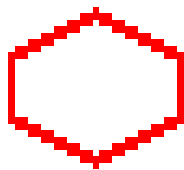
> *update()* (*component[1] = *TransformComponent*), this means we are calling *TransformComponent->update()*

3.2 TransformComponent : public Component

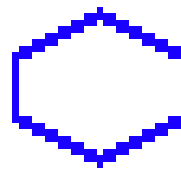
- Used to transform the position of the chicken
- We needed to use an extern variable *center_position* from *UCW.cpp* file, which calculate the center of the tile.
- We also used a *Vector2D*, called *position*. *Vector2D* contain many overloaded operator, which allow us to perform arithmetic operation like vector on the position.
- There is also the *posRow*, *posCol*, which store the coordinate of the chicken as (row,column).
- We use 3 overloading constructor: the first one will set the position to be second; the second one set the position as indicated by the parameters; the last one set the position but according to the top left corner of the rectangle.
- Void *getPosition()*: get the current position of the chicken but in row and column
 - First, we calculate the distance from the pixel coordinate of the chicken to the pixel coordinate of the center of every tiles (as specified in *SDL_Rect center_position*).
 - Then, we compare and see which is the shortest and we conclude that the chicken is staying in the row and column of the tile.

3.2 SpriteComponent : public Component

- Control the display (character image, range indicator ...) of each chicken.
- We also need to use the extern variable “position” (the top left corner of the render box of each tile) and “center position” defined in *UCW.cpp*.
- The “turn_indicator” and “range_indicator” are the textures representing the hexagon block indicating the player turn and range:



Turn indicator



Range indicator

- The *SpriteComponent::SpriteComponent()* constructor will call the *SpriteComponent::setTex()* function that load the texture from the specified path.
- The *SpriteComponent::init()* function load the *TransformComponent* of this chicken into the transform pointer. Then we create an instance of *Map*. Also, we need to specify the initial position

of the chicken's image.

- In *SpriteComponent::update()* function, we update the the position of the chicken's image according to its *TransformComponent::position*. However, since position.x and y are float, we need to cast it back to int to assign it to *destRect*.
- In *SpriteComponent::draw()*, we have to get the current position of the mouse to draw the range indicator. We do so by get the pixel position of it in *SDL_GetMouseState()* and calculate it (row,col) coordinates. We the checked whether this is the chicken's turn (*StatsComponent::myturn*), then we will draw the turn indicator at the tile the chicken is standing. Also, we checked whether the mouse is within the range of the chicken, then we will draw the range_indicator.

3.3 StatsComponent : public Component

- Purpose: Store the stats of each chicken
- In the construtor, all the stats will be initialized as specified in the parameters.

C. UI - User Interface

1. UI base class

- Render image and box to the screen and allow user to interact on it. Make use of polymorphism

```
protected:
    // Rectangle to draw box or background
    SDL_Rect src, dest;
    SDL_Rect bsrc, bdest;
    // Texture box and background (basic elements of UI)
    SDL_Texture* box, *background;
    // Color of text (if any)
    SDL_Color color[2];
```

to simplify the code.

- We create some features of UI that its children class may need as color, rectangle box to render or box and background. We also put it in protected so that these member can only be accessed in derived classes.
- Virtual int draw() and virtual void clean() is 2 basic functions of all of our derived classes so we use virtual to overload it in those child classes. draw() for rendering to the screen and clean() for destroy the texture that we have loaded.

1.1 Menu : public UI

- Use to display Menu whenever the game start or end.
- Derived from UI.
- We need to include “*TextureManager.h*” and “*LoadTextureFromText.h*” for rendering image (PNG file) and Text (string) purpose.
- We use bool selected[2] to check which button is selected (upper or lower button)
- In this menu, we use *LoadTextureFromText* to render the Game tile and 2 button option.
- We also set three color red, black and purple based on RGB for different purpose.
- We overload draw() function so that it received 2 string which is the name of 2 button (in our game is Start/Restart and Exit).
- In draw() function, at first we open some font to render the Title (purple colored) and 2 button (black colored). Then, we check the position of the mouse, if the mouse is located in one button region, the color of the button text will turn red, if not it will turn black as default. Moreover, if you clicked mouse on 1 button, it will return value to the *UCW::render()* to continue to other step or exit game. In addition, we also have a case for clicking X button on the name bar of game for exiting purpose.

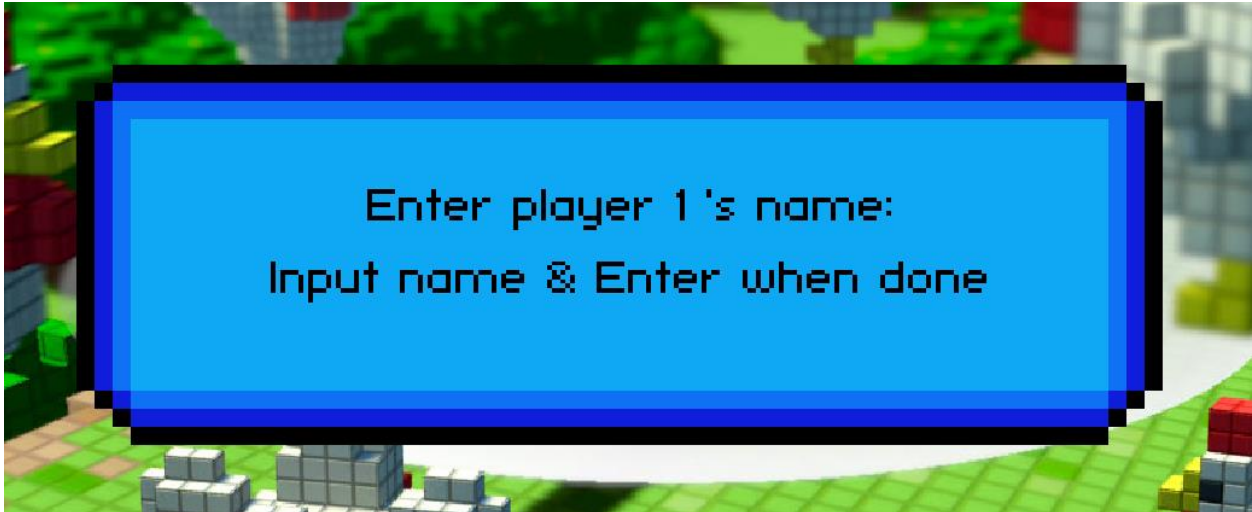
The result is:



1.2 NameInput : public UI

- This class is used to display 2 Name Input box after you clicked Start/Restart and then save 2 player name to Player_system.
- Derived from UI.
- We created 2 boolean variable to check if you have done the input section and if there any rendered text.
- There are also some temporary string for storing the name and guide text. In this class, we use *LoadTextureFromText* to render the box and text to the screen.
- In the constructor, it received the string from outside and use it as the default name (we use “player 1” and “player 2” as default), text color to black, and set the value of x,y,w,h for rendering the input box and background.
- draw() function is override to get the pointer of players from Player class for save name that have inputted to this instances. Similar to previous, we open font than draw background, box and some guide text on the screen and set name to be empty . In this function, if you enter some character in keyboard, it will be received and stored in name variable and then display it on the display. We also create case for backspace to delete a character or copy and paste. If there is any new character inputted, we will render it on the screen and after you finished by hitting the ENTER key, we will return 1 to UCW::render() for next step and save the name of player in Player class instance for further purpose.

- We override the `clean()` function to close box, background, font and texture that we have used and all the variable back to default in case of restart is called. The final result is:



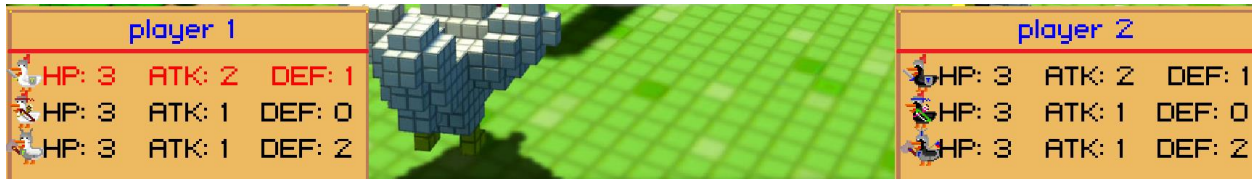
1.3 Popup : public UI

- This class is used to represent Popup box and info that you want to display (in our case is when attacking).
- Derived from UI.
- In this class, we create a string called `text` to store the information that you want to display.
- In the constructor, we defined the box rectangle location to render and save the received text to `text`. We also received height and width of the text.
- `Draw()` function is overridden, it will render the box and then the text such as “Attack or not” in black color. This will be the base for attacking function. When you click on the “attack” the chicken will hit others or “not” to do nothing. In our game, this Popup is called whenever you click on another chicken in attack range.
- `clean()` is override to close the font, text and box that we have rendered.

1.4 Stats_table: public UI

- This class is used to represent both player’s chicken stats such as health point (*hp*), defend (*def*) and attack (*atk*).
- Derived from UI.
- In this class, we set 3 different color for text consists of red (for dead or selected chicken), black (stats information color) and blue (player name). We also create 2 box on the left and on the right rectangle, location of each chicken stats and an image list of PNG pictures of chicken class and *LoadTextureFromText* for rendering stats text.
- In the constructor, we defined the location of each chicken image and 2 box.

- *Void Stats_table::draw()* function is overloaded to get 2 name player from Player class instances. Then we draw 2 box and 6 chicken and render all stats of chicken from stats_player class (from *StatsComponent*). In addition, if it is chicken 1 turn, it stats will turn red and similar to others turn. If the chicken is dead (hp = 0), instead of displaying stats, we will display a red colored “DEAD”. This function will render continuously until the game end.
- *Void Stats_table::clean()* function is override to close chicken image, font , box and text that be used.
- The result is:



1.5 Map : public UI

- This class is used to represent Map.
- Derived from UI.
- Has a friend relationship with function *UCW::handleEvents()* and class *SpriteComponent*.
- Included fstream for input from file purpose and stdlib.h for randomization purpose.
- This class has many member of *SDL_Texture* stand for each terrain and item we have, we also have create two 2D array of size 9x16 for map and *itemMap* which will be loaded from *.txt file in map_list and item_list. Last but not least is the random number for generating map randomly. The *.txt file is like:

```

2 3 0 0 0 0 4 9 1 1 0 0 0 0 0 0
2 3 0 0 0 0 4 7 8 9 1 0 0 3 0 0
2 3 0 0 0 0 5 6 6 4 0 0 3 3 0 0
2 3 0 0 0 0 0 4 5 0 0 0 3 3 0 0
2 2 2 2 2 2 4 4 4 0 0 0 3 0 0 0
2 2 2 2 2 2 4 9 4 4 0 0 0 0 0 0
4 1 4 4 4 4 9 7 8 4 0 0 1 1 0 1
4 4 4 4 4 7 6 6 8 9 4 0 0 1 1 1
5 4 4 5 9 7 6 5 5 6 8 5 0 3 1 1

```

- This class have friend class with *UCW* and *SpriteComponent* to allows these classes to access private instances such as map and item.
- *Void Map::LoadMap()* function is used to load Map and Item randomly based on a random number whenever we create a new instance. Then the map and item will be stored in 2 array map and *itemMap*.
- The constructor is used to load texture of terrains and items from PNG file, define a rectangle to draw each tile of terrain, generate a random number and call *void Map::LoadMap()*.



A hexagon tile sample that we have made by PISKEL

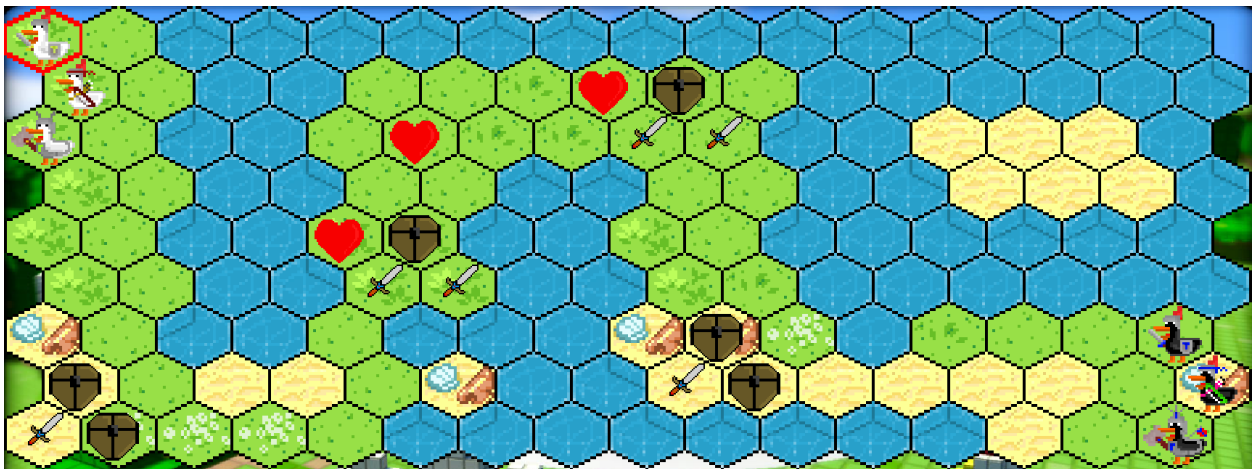
- The destructors is used to clear defined textures from computer memory.

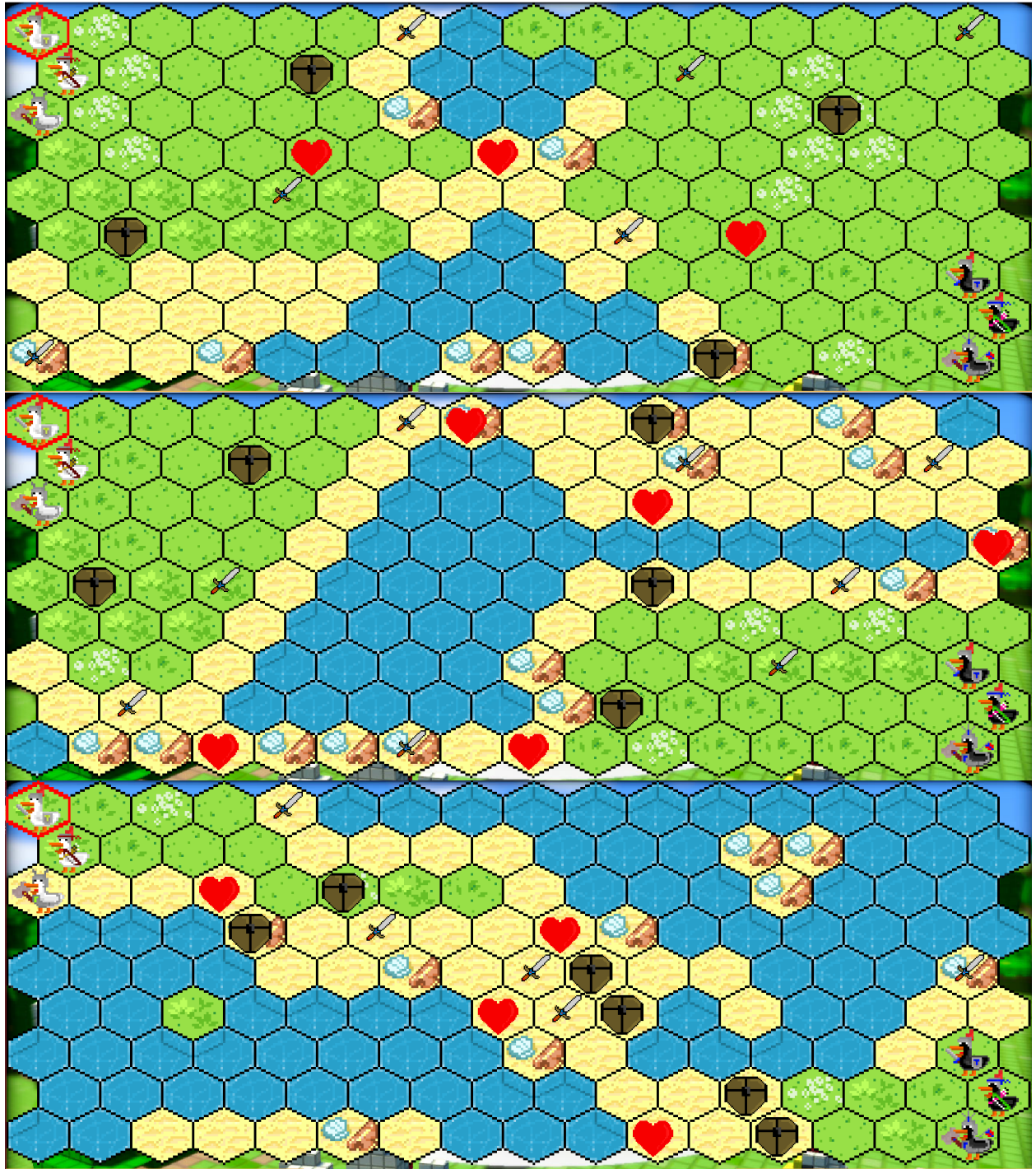
```

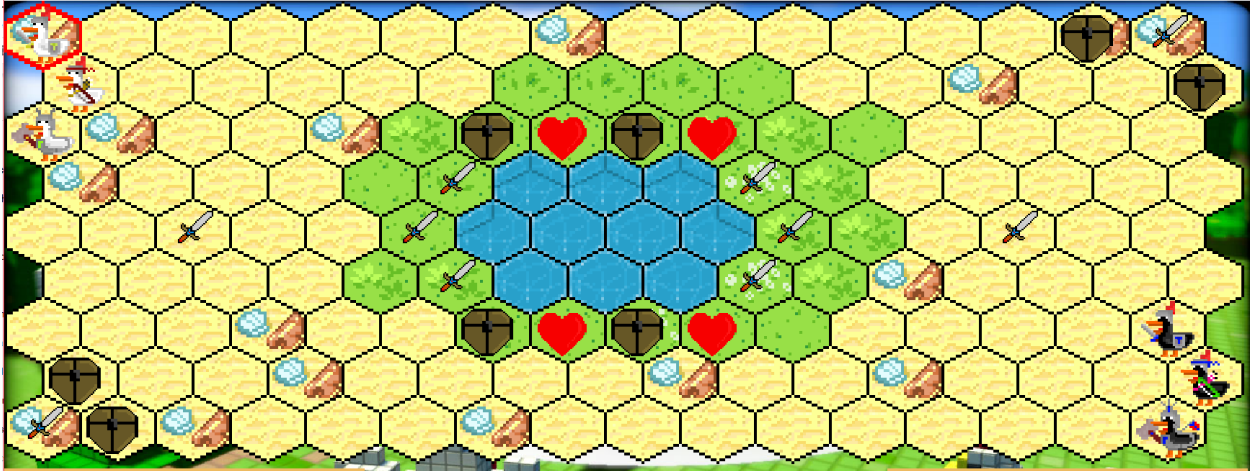
if (row % 2 == 0) {
    dest.x = column * 104 ;
    dest.y = row * 100 - 32 * row ;
    dest.w = 108 ;
    dest.h = 100 ;
}
else {
    dest.x = column * 104 + 52;
    dest.y = row * 100 - 32 * row ;
    dest.w = 108;
    dest.h = 100;
}
switch (type) {

```

- *Int Map::draw()* function is overridden to draw each tile of terrain (size 108x100) based on a for loop. We also create a formula for drawing hexagon map that we will shift row up 32 pixel ($\frac{1}{3}$ of 100), not included the first row (which is row 0) and if the row is even we will shift the column to the right 52 pixel. Then we use switch to draw each tile base on the map array value such as 0 stand for grass, 6 stand for water,etc. We will do similarly with the item. And the final result is some map with item and chicken on it that we have successfully generated:

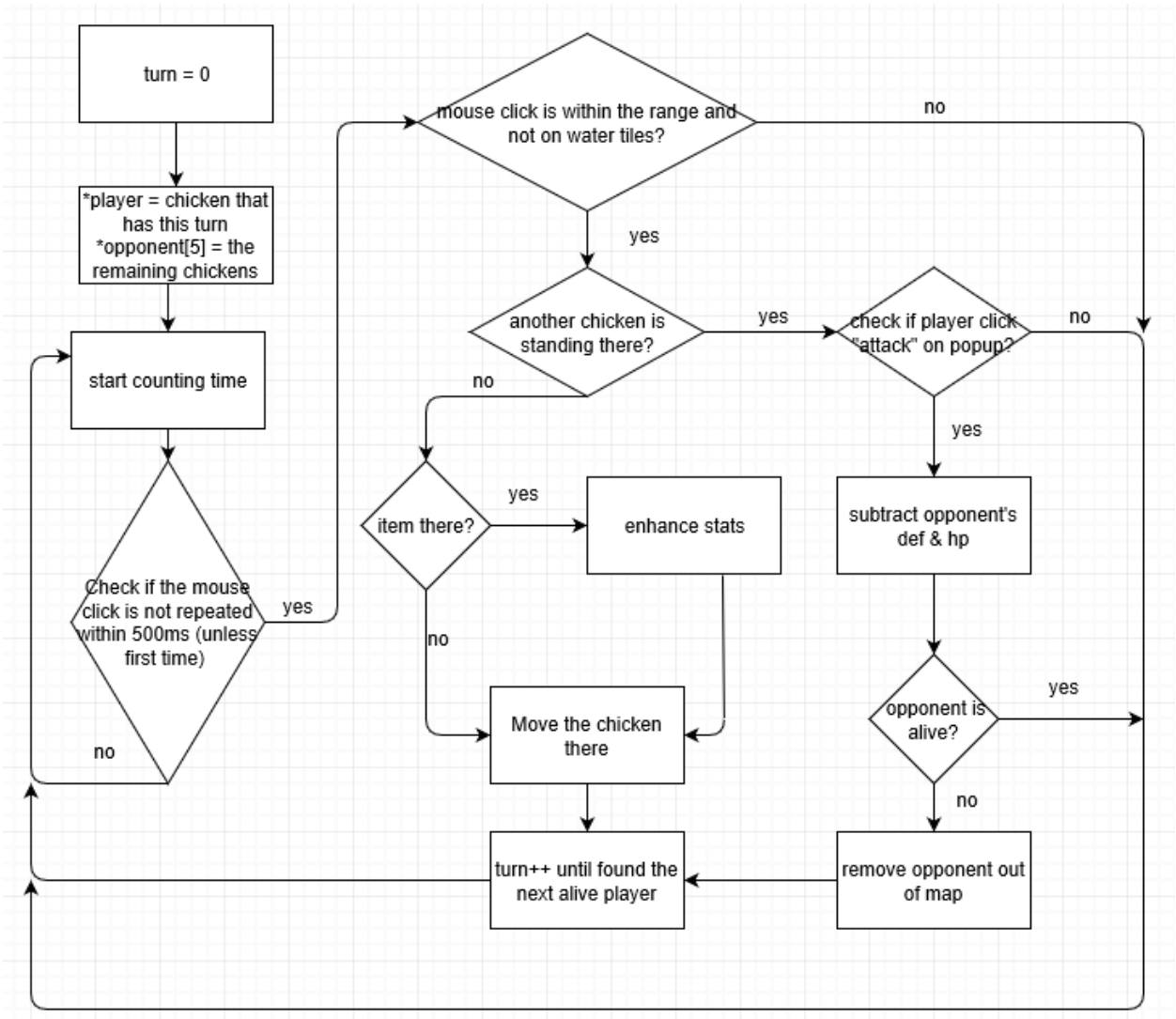






D. Turn-based algorithm - Mouse controller

- Whenever there is a mouse click, the `Mouse_Controller` is called() and this will handle the movement and turn counting of the game:
- We can easily get the pointer to each chicken by accessing the static variable `Player::allChickenList[6]`



E. Additional classes

All of below classes is using for load and render image to the screen which is mainly using SDL function which is outscope of our project so we will not say so deeply about it.

1. TextureManager

- This is an independent class which is created to render image to screen by using *SDL2*.
- The *SDL_Texture* TextureManager::LoadTexture(const char*)* will create a temporary surface to load image and create texture from this surface, then it will return the texture and free the surface.
- *Void TextureManager::Draw()* function will render the texture that we have loaded to the UCW::renderer which is our main renderer. Then it will be show on the screen by

SDL_RendererPresent.

2. LoadTextureFromText

- This class is also an independent class which have the same purpose as *TextureManager* but the target is text instead of image.
- The main idea of this function comes from LazyFoo tutorial of *SDL2*.
- We will have function that allows us to Loads image at specified path (*loadFromFile*) which is the same as *TextureManager::LoadTexture* or creates image from font string with color input (*LoadFromRenderedText*, we mainly use this function).
- The *free()* function will free texture if it exists.
- The *render()* will set rendering space and render it to the screen.
- The *getWidth()* and *getHeight()* is return the width and height of text, we use it for centering purpose.

3. Vector2D

- This class is used for *TransformComponent* and calculating on vector by defined a vector is a instance (x,y).
- This is one of our practice exercise that we have done in our tutorial class.
- Member x, y is the position of the object on 2D plan.
- We have 2 constructor, one allows you to define the x and y and one will make the default by set x,y equal to 0.
- Add, Subtract, Multiply, Divide is calculating function in vector and then we use it to overload operator +, -, *, /. This really help us to simplify our works.
- *Vector2D::Zero()* function to set the position back to origin.
- The last operator overloading is << of output stream to print to screen to coordinate of the point when we test and debug our code on console.

VI. Self-evaluation

By working on this project, we have gained lots of hands-on experiences with object oriented programming, C++ language, graphic aiding library like *SDL2*, game system and designing concept.

We concluded that we have basically achieved most of the goals we set out in the proposal. However, we have not achieved the goal of designing an AI to play with the player due to the complexity of the problem. We might figure it out in the future.