C++ Capstone Project Rubric

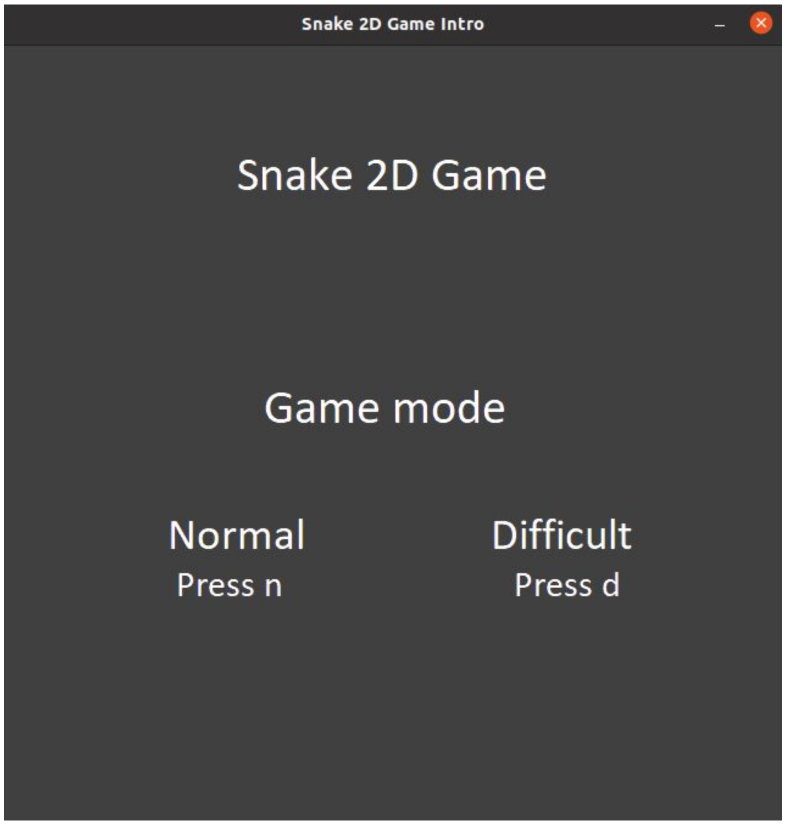
Chosen Project

I chose Capstone Option 2: The 2D Snake game Github repository is here.

New Features:

1. GameIntro window

Upon running the game, a game intro window appears to ask the user if he will play in **normal** game mode or **difficult** game mode as follows:



This feature is implemented in the renderer class using the function

```
GameMode Renderer::RenderGameIntro(const std::size_t screen_width, const
std::size_t screen_height, const char* imageFileName)
```

2. Game mode

In the previous window the user should press either n for normal game mode or d for difficult game mode. The default mode if any other key is pressed is the normal mode.

Game Mode	Normal	Difficult
Snake start speed	0.1	0.2
Obstacles	No Obstacles	Add Obstacles

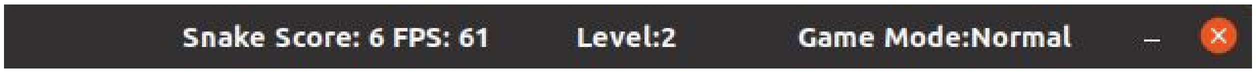
This game mode is set based on the previous window and passed to the Game constructor as follows:

```
Game::Game(std::size_t grid_width, std::size_t grid_height, GameMode mode)
   : _mode(mode),         // Feat: Initialize the _mode private member
...
...
```

3. Update the game title with level and mode

Level value is calculated based on the score (in the current implementation the level increased every 3 points) and updated in the game title in the Game::Run function as follows:

```
   // Feat: set the level based on the score and the levelScore constant
   if((score % levelScore) == 0) { level = score / levelScore; }
   // Feat: update title with the score, fps, level and game mode values
   renderer.UpdateWindowTitle(score, fps, level, _mode);
```

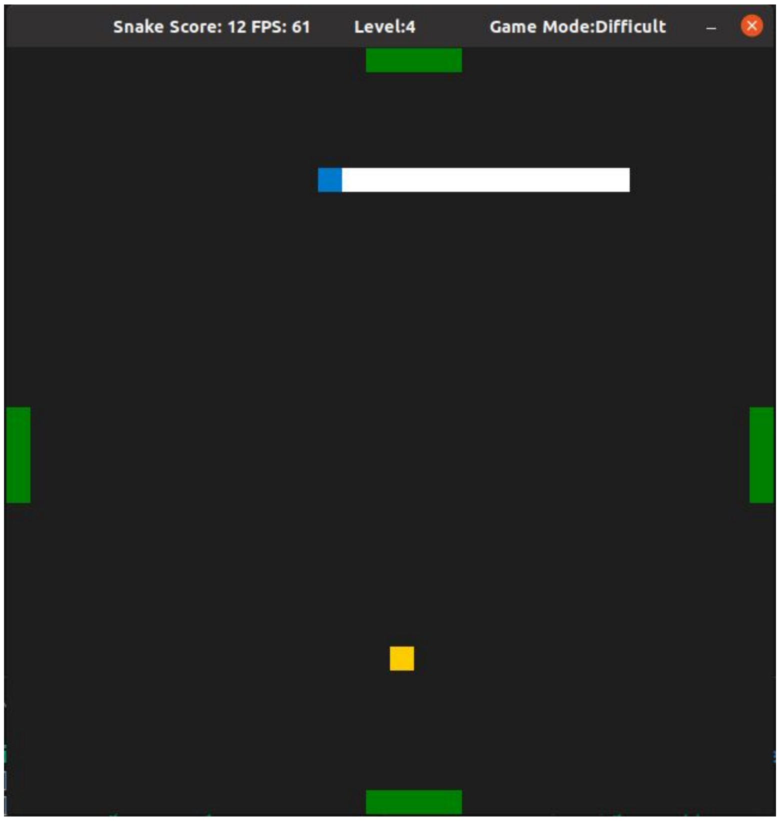

Snake Score: 6 FPS: 61 Level:2 Game Mode:Normal

4. Obstacles

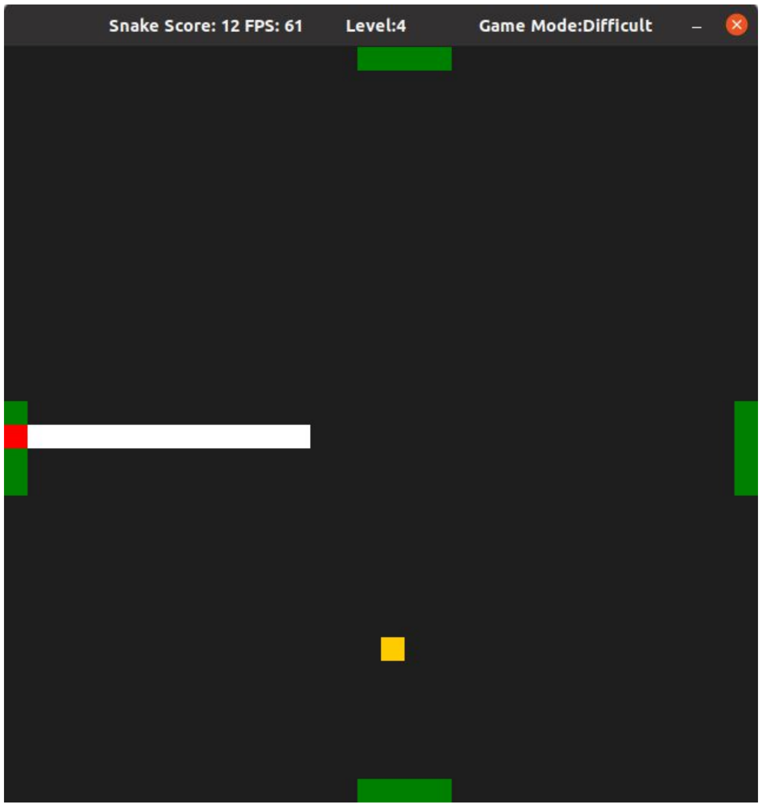
Obstacle class is added as follows:

```cpp
#ifndef OBSTACLE_H
#define OBSTACLE_H

#include <vector>
#include "SDL.h"

// Feat: add enum to differentiate between 4 obstacles in the game
enum ObstacleNumber{
   NoObstacle=0,
   Obstacle_1,
   Obstacle_2,
   Obstacle_3,
   Obstacle_4
};

// Feat: add class Obstacle to encapsulate the functionality of adding
obstacles in case of difficult game mode
class Obstacle{

public:
   // Feat: constructor
   Obstacle(int gridWidth, int gridHeight);
   // Feat: destructor
   ~Obstacle(){};
   // Feat: get the obstacle body points
   std::vector<SDL_Point> GetObstacleBody() const;
   // Feat: getter/setter to obstacles count private member
   int  GetObstacleCount();
   void SetObstacleCount(const int obstacleCount);
   // Feat: add an obstacle
   void CreateObstacle();
   // Feat: Remove the obstacles body and reset the obstacles count
   void RemoveObstacle();
   // Feat: check if the cell is occupied by an obstacle
   bool ObstacleCell(int x, int y);
   static const int MAX_OBSTACLE_COUNT{Obstacle_4};

private:
   // Feat: 1D vector of obstacle body grid points
   std::vector<SDL_Point> _obstacleBody;
   int _gridWidth;
   int _gridHeight;
   // Feat: number of obstacles added
   int _obstacleCount{NoObstacle};
   // Feat: obstacle body is of length 4 cells
   const int _ObstacleLength{4};
   // Feat: 2D vector that will hold the positions of the 4 obstacles on the
grid
   std::vector<std::vector<SDL_Point>> initialObstaclePoints;
};
#endif /*OBSTACLE_H*/
```

If the Game mode is chosen to be difficult then there are 4 obstacles that will be added each after (3 points/ 1 level) and when the score reaches multiple of 15 points (level 5,10,...), the obstacles are removed and then are added again one by one after every 3 points.
The 4 obstacles could be shown as follows:



If the snake run into any of the obstacles will die:



An Obstacle class instance is added as a smart pointer private member of the Game class as follows:

```
// Feat: add private member _obstacle using smart pointer
 std::unique_ptr<Obstacle> _obstacle;
```

In the function Game::PlaceFood, the location of the generated food is checked that it is not occupied by any obstacle as well as the snake body as follows:

```
// Feat: check also that the location of food is not occupied by any obstacle
   if (!snake.SnakeCell(x, y) && !_obstacle->ObstacleCell(x,y)) {
     food.x = x;
     food.y = y;
     return;
   }
```

The check if the snake run into any of the obstacles are done in the Game::Update function as follows:

```
// Feat: check if the snake run into an obstacle
 if(_obstacle->ObstacleCell(new_x,new_y)){
   snake.alive = false;
   return;
 }
```

The logic of creating/removing obstacles is implemented in the Game::Update function as follows:

```
// Feat: check if the game mode is difficult and the max obstacle count is not
reached to add obstacle every level
   if((_mode == GameMode::Difficult) &&
_obstacle->GetObstacleCount()!=Obstacle::MAX_OBSTACLE_COUNT &&
score%levelScore==0)
   {
       _obstacle->CreateObstacle();
   }

   // Feat: after every 15 points reset/remove the obstacles
   if(score % 15 == 0)
     _obstacle->RemoveObstacle();
```

The rendering of the obstacles is done in the Renderer::Render function as it has a const obstacle passed to it by reference as follows:

```
void Renderer::Render(Snake const snake, SDL_Point const &food, Obstacle const
&obstacle) {
 SDL_Rect block;
 block.w = screen_width / grid_width;
 block.h = screen_height / grid_height;

 // Clear screen
 SDL_SetRenderDrawColor(sdl_renderer, 0x1E, 0x1E, 0x1E, 0xFF);
 SDL_RenderClear(sdl_renderer);

 // Feat: Render Obstacles (color: green) based on the obstacle body points
 SDL_SetRenderDrawColor(sdl_renderer, 0x00, 0x80, 0x00, 0xFF);
 for (SDL_Point const &point : obstacle.GetObstacleBody()) {
   block.x = point.x * block.w;
   block.y = point.y * block.h;
   SDL_RenderFillRect(sdl_renderer, &block);
 }
```

5. Press Escape to exit the game at any time

Check on the Escape key press event is added to the Controller::HandleInput function as follows:

```
// Feat: Exit the game on pressing Escape
       case SDLK_ESCAPE:
         running = false;
         break;
```

6. Fix a problem with placing the food

The random generator values for food x and y points were in the range (0, grid_width) and (0, grid_height) and this was leading to sometimes generating the food location outside of the configured window and the modification needed was just to make the upper bound of the ranges as grid_width-1 and grid_height-1 as follows in the Game class constructor:

```
 random_w(0, static_cast<int>(grid_width-1)),    // Fix: initially grid_width
used not (grid_width-1) lead to out of boundary food point
 random_h(0, static_cast<int>(grid_height-1)){   // Fix: initially grid_height
used not (grid_height-1) lead to out of boundary food point
```

The Covered Rubric Points

1. The project demonstrates an understanding of C++ functions and control structures.

Addressed in almost all the above features through using of functions, loops, enum, … etc.

2. The project accepts user input and processes the input.

Mainly addressed in the Game intro window that let the user choose the game mode based on pressing a certain key.
Function: Renderer::RenderGameIntro

3. The project uses Object Oriented Programming techniques.

   Addressed in Obstacle class
   Files: obstacle.cpp, obstacle.h

4. Classes use appropriate access specifiers for class members.

   Addressed by using private/public members in the Obstacle class (**obstacle.h**)

5. Class constructors utilize member initialization lists.

   Addressed by using the initialization lists in Obstacle class constructor (**obstacle.cpp**) and modifying Game constructor to initialize _mode, _obstacle through initialization list

6. Classes abstract implementation details from their interfaces.

   Mainly Addressed by the **Obstacle class**

7. The project makes use of references in function declarations.

   Addressed by passing the GameMode variable mode by reference to the **Game constructor** and passing the obstacle by reference to the **Renderer::Render** function

8. The project uses smart pointers instead of raw pointers.

   Addressed once by adding an obstacle as a unique pointer private member to the **Game class**

Note: All the modified code/features are preceded with the comment [// Feat:] followed by a brief explanation of the code added.