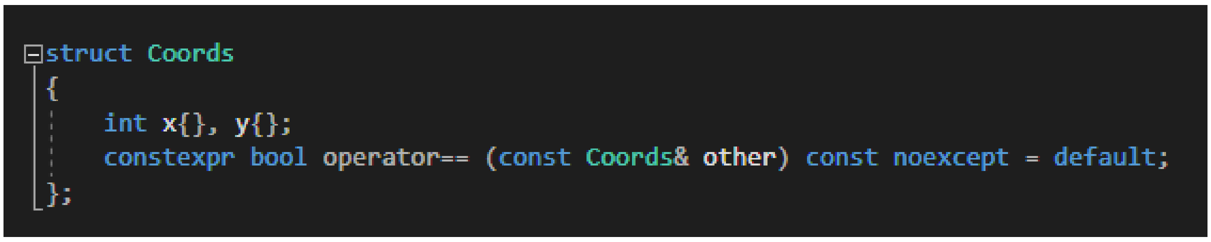
API Design: Snake Re-Refactor Report

Daniel Burt

The first thing I did was read the critique I had received as a checklist. Most of these were fairly low-hanging fruit which could be addressed quickly.

- I removed all redundant files, as per the instructions.
  Collider
  Image
  Sprite
- I enforced the rule of zero everywhere it made sense. This was applicable to every struct comprised of basic types.
  Rectangle
  Color
  Vector2
  Coords

  Ex:



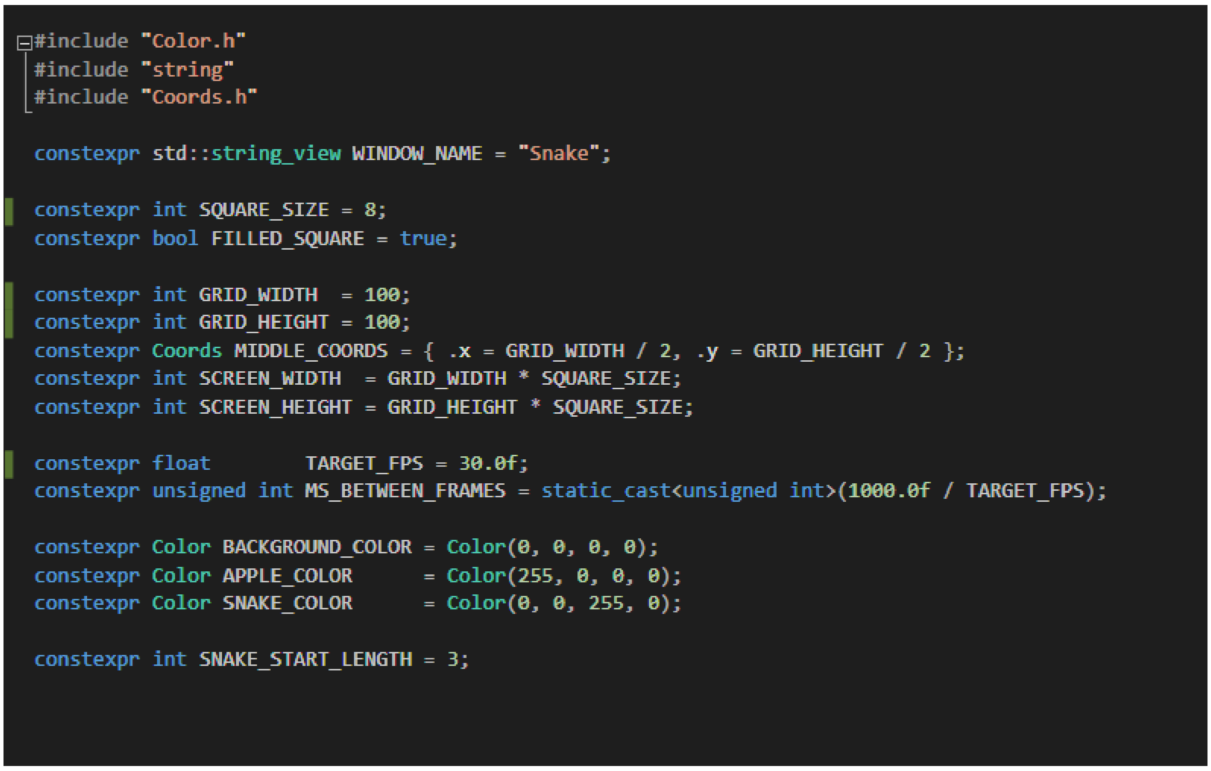
```
struct Coords
{
    int x{}, y{};
    constexpr bool operator== (const Coords& other) const noexcept = default;
};
```

- I undid some unnecessary casting
- I added appropriate specifiers where they were missing
  Snake
  Vector2

I continued by making some larger changes, some of which were inspired by the codebases of my peers and others I remember from the code analysis in class.

I made a header file called Constants.h. I got this idea from Filippo's code. This contains pretty much every unchanging variable in the game you may feasibly want to configure. This means I can completely remove a lot of stuff:

- Grid
  Since the grid only really contains unchanging values I can just store some constants instead. Both of its useful functions were moved to Game
- Apple
  Apple had a Coord and a Color. Since I can just make the Color a constexpr the apple only holds a Coord, so I just replace it with a Coord.
- Config
  The config struct would hold some constants that I would use to initialize the window and things. Now I just read from my list of Constants when I initialize the program.



```cpp
#include "Color.h"
#include "string"
#include "Coords.h"

constexpr std::string_view WINDOW_NAME = "Snake";

constexpr int SQUARE_SIZE = 8;
constexpr bool FILLED_SQUARE = true;

constexpr int GRID_WIDTH  = 100;
constexpr int GRID_HEIGHT = 100;
constexpr Coords MIDDLE_COORDS = { .x = GRID_WIDTH / 2, .y = GRID_HEIGHT / 2 };
constexpr int SCREEN_WIDTH  = GRID_WIDTH * SQUARE_SIZE;
constexpr int SCREEN_HEIGHT = GRID_HEIGHT * SQUARE_SIZE;

constexpr float        TARGET_FPS = 30.0f;
constexpr unsigned int MS_BETWEEN_FRAMES = static_cast<unsigned int>(1000.0f / TARGET_FPS);

constexpr Color BACKGROUND_COLOR = Color(0, 0, 0, 0);
constexpr Color APPLE_COLOR      = Color(255, 0, 0, 0);
constexpr Color SNAKE_COLOR      = Color(0, 0, 255, 0);

constexpr int SNAKE_START_LENGTH = 3;
```

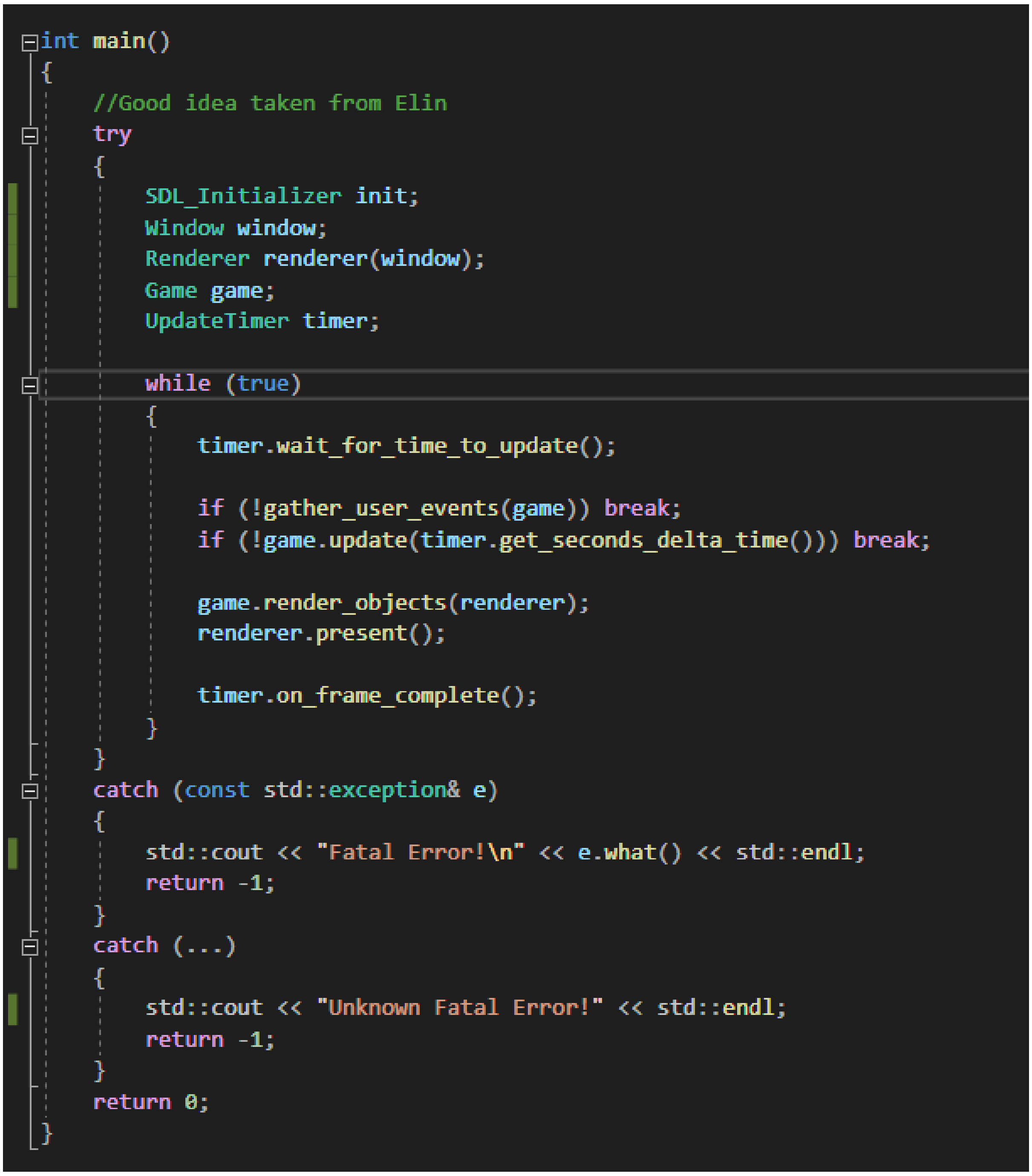
I removed Transform and Vector2. They're only used for rendering and the RenderManager's days were numbered.

Next I handled the SDL implementation. My last approach was to stick the pointers in a struct and initialize them there. This violates the principle that an object should handle no more than one resource.

What I did instead was create three structs which handle SDL resources. I drew much inspiration from Elin's code.

- SDL_Initializer holds nothing but calls SDL_Init in its constructor and SDL_Quit in its destructor. Throws an exception if SDL_Init returns a negative number, as it means that it failed.
- Window holds an SDL_Window*. It's created in the constructor and freed in the destructor. Throws an exception if creation fails.
- Renderer holds an SDL_Renderer*. Its constructor takes a Window& which it uses to create the SDL_Renderer. Throws an exception if creation fails.

I moved all of main into a big try-catch-block so we abort if anything weird happens. I declare instances of these three structs in the order that they are dependent upon each other in main.

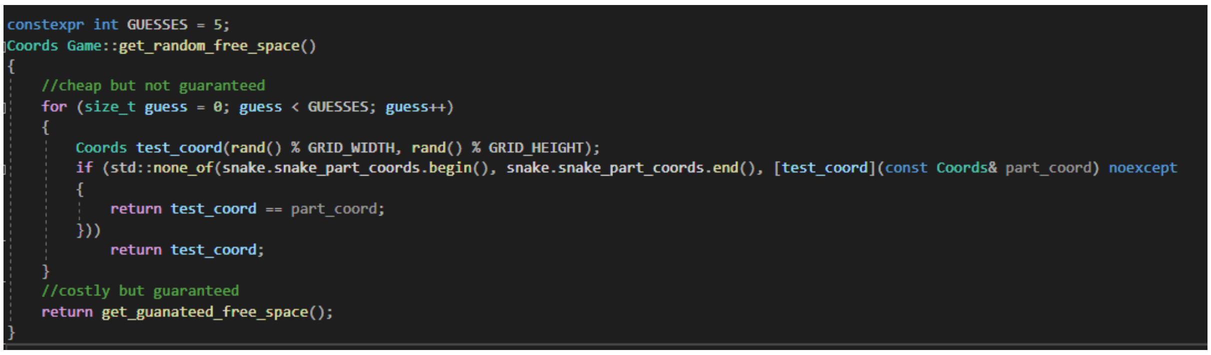


```cpp
int main()
{
    //Good idea taken from Elin
    try
    {
        SDL_Initializer init;
        Window window;
        Renderer renderer(window);
        Game game;
        UpdateTimer timer;

        while (true)
        {
            timer.wait_for_time_to_update();

            if (!gather_user_events(game)) break;
            if (!game.update(timer.get_seconds_delta_time())) break;

            game.render_objects(renderer);
            renderer.present();

            timer.on_frame_complete();
        }
    }
    catch (const std::exception& e)
    {
        std::cout << "Fatal Error!\n" << e.what() << std::endl;
        return -1;
    }
    catch (...)
    {
        std::cout << "Unknown Fatal Error!" << std::endl;
        return -1;
    }
    return 0;
}
```

I decided to turn the aforementioned Renderer object into a replacement for the RenderManager. I kept the render-queue approach but made the RectEntries hold copies instead of references so it'd always work even if the objects render go out of scope (which they do). I also made RectEntries just render the Rectangles as they are given instead of using a Transform.

Another sore point was the inefficiency of choosing a new square for the apple. My approach was to make a vector of every available square and choose one at random. With a big play-area this can be super inefficient and slow when just guessing a random square will work almost all the time. However, it is always guaranteed to work when there is at least one available square. Choosing a random square until you hit an unoccupied one will work *almost* all the time, but the same guarantees are absent.

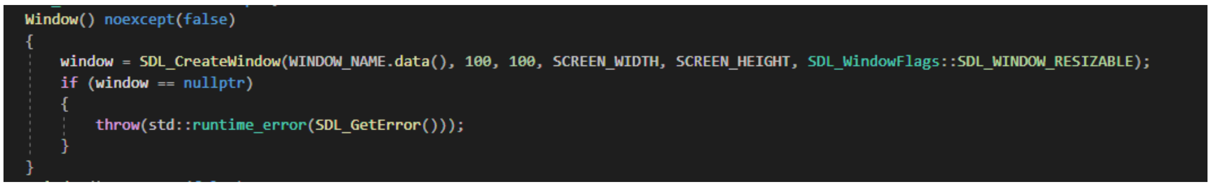
I didn’t want to compromise the fact that my algorithm is guaranteed to work so I chose to combine the two available solutions:



```
constexpr int GUESSES = 5;
Coords Game::get_random_free_space()
{
    //cheap but not guaranteed
    for (size_t guess = 0; guess < GUESSES; guess++)
    {
        Coords test_coord(rand() % GRID_WIDTH, rand() % GRID_HEIGHT);
        if (std::none_of(snake.snake_part_coords.begin(), snake.snake_part_coords.end(), [test_coord](const Coords& part_coord) noexcept
        {
            return test_coord == part_coord;
        }))
            return test_coord;
    }
    //costly but guaranteed
    return get_guanateed_free_space();
}
```

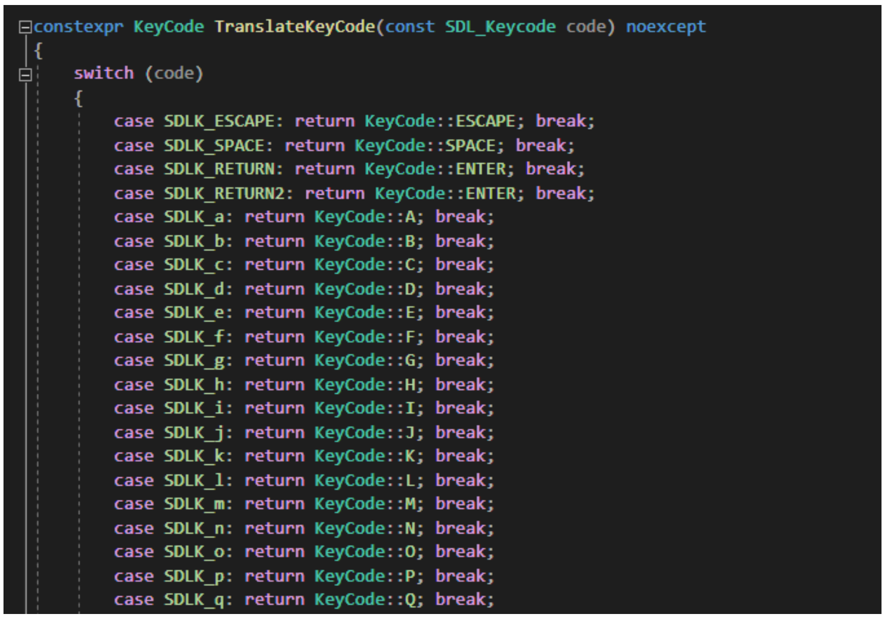
Now the program picks 5 squares at random, and if none of them are free, we make the vector of every square and so forth. The probability of picking five occupied squares in a row on a big board is vanishingly unlikely unless our snake actually occupies almost all squares, in which case the inefficient solution is most likely necessary.

One of the last thing I learned and implemented was the specifier noexcept(false), which explicitly states that a destructor can throw. It’s nice to be explicit and this got rid of a few invalid warnings complaining that my throwing constructor weren’t marked noexcept.



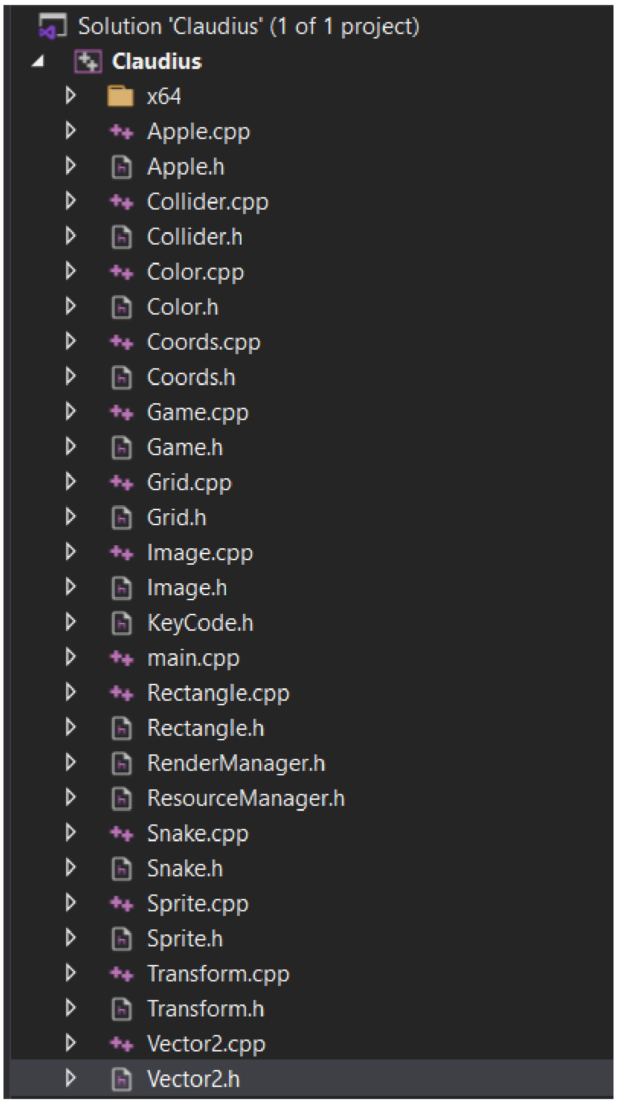
```
Window() noexcept(false)
{
    window = SDL_CreateWindow(WINDOW_NAME.data(), 100, 100, SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WindowFlags::SDL_WINDOW_RESIZABLE);
    if (window == nullptr)
    {
        throw(std::runtime_error(SDL_GetError()));
    }
}
```

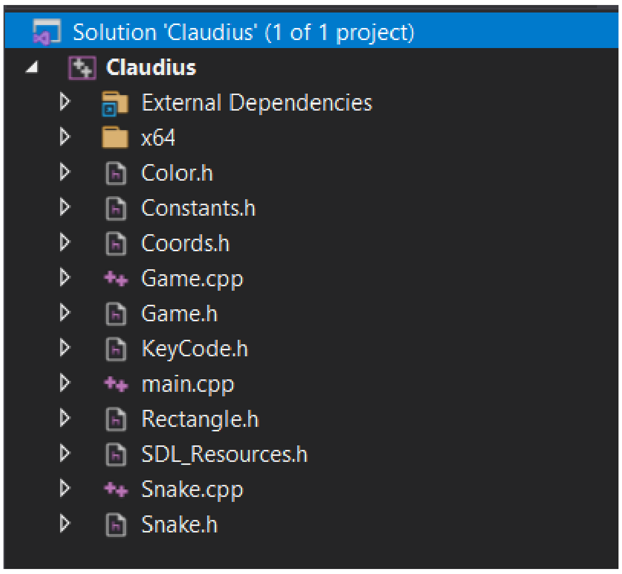
The final tiny optimization I made was make TranslateKeyCode constexpr for efficiency reasons



```
constexpr KeyCode TranslateKeyCode(const SDL_Keycode code) noexcept
{
    switch (code)
    {
        case SDLK_ESCAPE: return KeyCode::ESCAPE; break;
        case SDLK_SPACE: return KeyCode::SPACE; break;
        case SDLK_RETURN: return KeyCode::ENTER; break;
        case SDLK_RETURN2: return KeyCode::ENTER; break;
        case SDLK_a: return KeyCode::A; break;
        case SDLK_b: return KeyCode::B; break;
        case SDLK_c: return KeyCode::C; break;
        case SDLK_d: return KeyCode::D; break;
        case SDLK_e: return KeyCode::E; break;
        case SDLK_f: return KeyCode::F; break;
        case SDLK_g: return KeyCode::G; break;
        case SDLK_h: return KeyCode::H; break;
        case SDLK_i: return KeyCode::I; break;
        case SDLK_j: return KeyCode::J; break;
        case SDLK_k: return KeyCode::K; break;
        case SDLK_l: return KeyCode::L; break;
        case SDLK_m: return KeyCode::M; break;
        case SDLK_n: return KeyCode::N; break;
        case SDLK_o: return KeyCode::O; break;
        case SDLK_p: return KeyCode::P; break;
        case SDLK_q: return KeyCode::Q; break;
```

All in all, over these two refactorings I've removed almost 2/3 of the source files (the original year 1 project had 27 source files, and I've landed on 10).


Solution 'Claudius' (1 of 1 project)
Claudius
x64
Apple.cpp
Apple.h
Collider.cpp
Collider.h
Color.cpp
Color.h
Coords.cpp
Coords.h
Game.cpp
Game.h
Grid.cpp
Grid.h
Image.cpp
Image.h
KeyCode.h
main.cpp
Rectangle.cpp
Rectangle.h
RenderManager.h
ResourceManager.h
Snake.cpp
Snake.h
Sprite.cpp
Sprite.h
Transform.cpp
Transform.h
Vector2.cpp
Vector2.h


Solution 'Claudius' (1 of 1 project)
Claudius
External Dependencies
x64
Color.h
Constants.h
Coords.h
Game.cpp
Game.h
KeyCode.h
main.cpp
Rectangle.h
SDL_Resources.h
Snake.cpp
Snake.h

I believe this code has been boiled down to its essentials about as far as one would want to go, and is overall a much cleaner and more sensible implementation.