Title: Modern Game Development in Rust, a Comparison

Submitted by: Thomas Sieben

Murphy class: 2019

Submitted to: Prof. Pablo Durango-Cohen

Project adviser: Prof. Jesse Tov

Motivation:

This project will be evaluating the development cycle and performance of crafting video games (2D only) within several mediums. The language of most interest (and what sparked initial interest in this project) is Rust, a relatively new language[1]. It introduces many desirable features for a performant programming language, including a robust type system, efficient and controlled memory management, and support for concurrency and multithreading. In other words, it is safe (preventing compile or runtime errors or other unknown behavior) and fast (actual compilation time and execution). Additionally, the community support for Rust is rather fervent and widespread. For the past three years, it placed first in Stack Overflow's "Most Loved Programming Language" survey. Game development with Rust is also just in its beginning stages. Therefore, it would be desirable to be able to use this language as a tool in the game development process.

For comparison, it would be useful to create the same game that will be created in Rust with another programming language, this time C++. The reason for doing so is that, written idiomatically (writing code within the acceptable style or standards for a given language), C++ performs similarly to Rust. However, C++ can be unintuitive or run into issues that Rust has no trouble with, including the aforementioned concerns of type safety and control of memory. Additionally, Rust and C++ are

[1] The first stable release of Rust 1.0 was on May 15, 2015. (The Rust Core Team. May 15, 2015. "Announcing Rust 1.0".)

similar enough where it would be useful to benchmark performance directly between the two languages. This could provide a good framework for understanding performance for creating the same game in two different languages.

Finally, a third format for creating the game that the previous two languages implement will be used, as this third format is the most common game development tool in the industry. The tool is the Unity game engine, a framework that does most of the heavy lifting for the developer, including support for building games on modern consoles (such as PlayStation, iOS, or Windows), a physics engine, graphics rendering, etc. There is really only one major scripting API that is currently used with Unity, that being C#, so this same game will be created using this method.

This project aims to explore the current methods of game development available through three mediums: Rust, C++, and Unity/C#. One of the goals is to discover what is currently available for developing on Rust (through what are called "crates," or libraries of code that can be imported into one's project—these crates that have been created by the Rust community, for example, to render graphics or simulate physics). Another aim of this project is more exploratory in the sense that not many games have been developed within the Rust ecosystem (see http://arewegameyet.com/index.html#games). This site, which provides references for the Rust community on gaming resources for developers, aptly states "Are we game yet? Almost. We have the blocks, bring your own glue." This project will explore the level of difficulty with which a game can be "glued" together, and more importantly, whether that effort is worth it compared to an established platform in Unity. While C++ is not extremely common for creating games, it is still widely used simply due to its ubiquity and performance. Additionally, it makes a good basis for comparison for Rust.

Research:

Many game engines were considered for this project from the Rust side of things. The three core engines that seem to have the most downloads and support were piston, amethyst, and rust-sdl2. As of writing for this report, piston had 122,000 downloads, amethyst 10,000, and rust-sdl2 149,000 (http://arewegameyet.com/categories/engines.html). This project explored the use of rust-sdl2, as it had both the most downloads over the other two, as well as a fundamental similarity to the other engine used for C++. The C++ engine, ge211, was designed by the adviser to this project (for use in teaching Fundamentals of Programming II at Northwestern University, to create games), and uses sdl2 libraries and bindings. SDL2 is an open-source cross-platform software development library which provides a hardware abstraction layer for different components, e.g. low-level access to audio, keyboard, mouse, etc. SDL is known as Simple DirectMedia Layer. (https://www.libsdl.org/). SDL is written in C and works natively with C++, while the rust-sdl2 crate/engine used for this project provides bindings to use the software.

The reason it was useful to design a game using both ge211 and rust-sdl2 was that since both used the same SDL2 software, the comparison between the two's performance would be much simpler. More research could be done when looking at piston and amethyst, but since they were less adopted than rust-sdl2, they were not considered. There are so many more game engines for Rust than just the three mentioned. There is support for the language, but without widespread adoption, these choices will never be as viable as something like Unity and C#, which has industry-wide adoption.

Design:

Now after selecting the engines to use, a game needed to be created. For this project, a simple snake game was designed, as it met the requirements of comparison. It's a 2D game, so simple enough for a hobbyist to create, and it can feasibly be designed in all three languages and

tested. In the appendix is attached the main functions for the snake game in ge211 and rust-sdl2. They can also be found at https://github.com/ths13/simple_games_ge211 and https://github.com/ths13/simple_games_rust_sdl2. The game was first designed using the ge211 engine. This engine, which uses the sdl2 libraries, has useful built-in functions that allow a developer to create a game fairly well after reading up on the documentation. Figure 1 below shows the working version of the snake game.

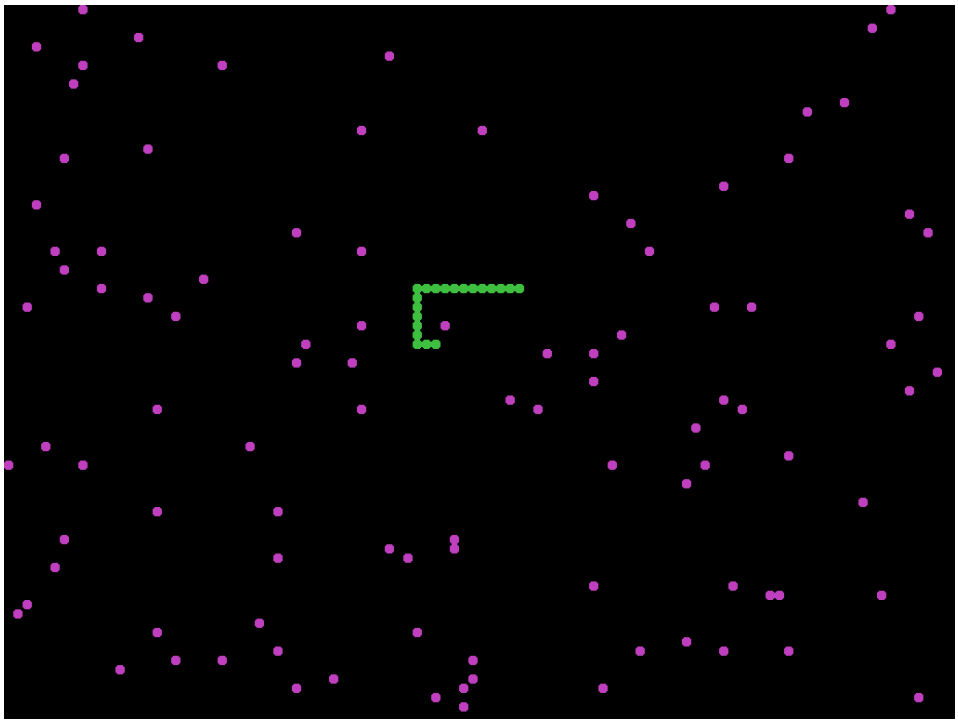

Figure 1: Snake game in ge211

To give a brief overview of how the game was constructed, it utilized five different structures, the Simple_snake struct, which inherits from the Abstract_game class, a View struct, which contains sprite definitions and the grid quantization methods (to produce discrete grid for the snake to move around), a Model struct containing the game functions and objects, and the Snake and Food structs, the game objects in this world. The code can be viewed on GitHub or in the Appendix.

The next language that was used was Rust. Using rust-sdl2, the same type of game was created. This facilitated a way to compare the two languages. The main difference with the Rust version was that there was no "update" method or built-in game loop. Instead, there is a main loop of sorts which takes in events from an "event_pump," responding to things like key presses. Approximately the same amount of lines were used to create the game with Rust versus C++. Finally the snake game was created with C# in Unity. This was the simplest of the languages, as Unity has a built-in editor which allows a user to view the scene without actually writing much code. In fact, a lot of the functionality can be implemented through the Unity editor. Some simple scripting in C# gets the game up and running.

Comparison:

Between the three languages, they all have their unique aspects and benefits. For this project, further games could have been developed which may have shown off the relative benefits of each language, but instead a simpler game was designed to allow for a more controlled environment and easier comparison. Designing a more graphically intensive game may have challenged the performance of each language more, but it is hard to say right now which would be most suitable for a graphically intensive design. Further discussion of this will follow.

In terms of why one would want to develop in any of these languages, this is a quick breakdown of each. Unity and C# is the gold standard. It is relatively straightforward to pickup and learn. It has been adopted by industry and is pretty much mainstream. The reason that it is such a go-to is that even non-programmers can learn how to use Unity to an extent, as much of the game design can be accomplished within the Unity editor and without scripting. However, scripting makes it easy to provide new functionality to games designed in Unity. Unity also uses a component-based system over standard Object Oriented Programming style. What this means is that instead of having

multiple classes which inherit from one another, say an enemy abstract class with an orc class and troll class inheriting from it, and possible further subclasses of orc, we instead have different game objects. So if I want an orc, I design a game object which has different components attached to it that make it an orc. I can give it different scripts as components (say, a script to control orc behavior), audio, meshes, colliders, etc. So mixing and matching is straightforward. This is also useful for things like serialization (how objects are saved and loaded in the scene and into memory). This allows novices and hobbyists to pretty quickly get a game up and running, and was relatively simple to create a snake game for.

Rust and C++ are the more similar of the three. Both don't have any industry-wide standardized game engine. For C++, this project used ge211, and for Rust, the rust-sdl2 crate. Both engines used sdl2 libraries, which made for easy comparison in terms of framerate and performance. For future work, it will also allow more granular testing of different functions if one is interested in learning more about performance. Also, both Rust and C++ used the more standard OOP (object oriented programming) style, with structures, classes, and inheritance to design their APIs (application program interfaces). In terms of the final results of how the two games compared, it was actually more straightforward to design a game with the ge211 engine. Of course, there may be bias depending on which language a user is more familiar with, but after spending time with both languages, there is just more ongoing support and resources available for games designed in C++. This is also a testament to how well designed the ge211 engine is. My adviser, Jesse Tov, created this, and the documentation and examples are straightforward.

If you are a hobbyist looking for a new language to pick up, want to familiarize yourself more with Rust, or are interested in creating games and comparing the process to C++ or something similar, then creating a game designed in Rust might be right for you. Outside of simple games (2D ones mostly), a general learning exercise, or teaching yourself a new language, Rust is

mostly suitable at the hobbyist level. There are only a few games created by other enthusiasts using the language, and maybe one or two at the production/industry level. I think this topic deserves much further exploration, as only one game was designed between the three languages. Surely someone who is within the industry or more an expert on Rust could provide more insight or contributions to this research as well.

Future considerations:

To further improve this project, the addition of microtesting, or adding some sort of performance comparison between Rust and C++, would be of use to those looking for a reason to switch to something that is faster. Additionally, using a more graphically intensive game might make this comparison even clearer. Prof. Tov, the adviser to this project, created a simple game, *fireworks*, that provided a rudimentary graphical interface for analyzing framerate of a game which has more animations to it. Figure 2 depicts this game.

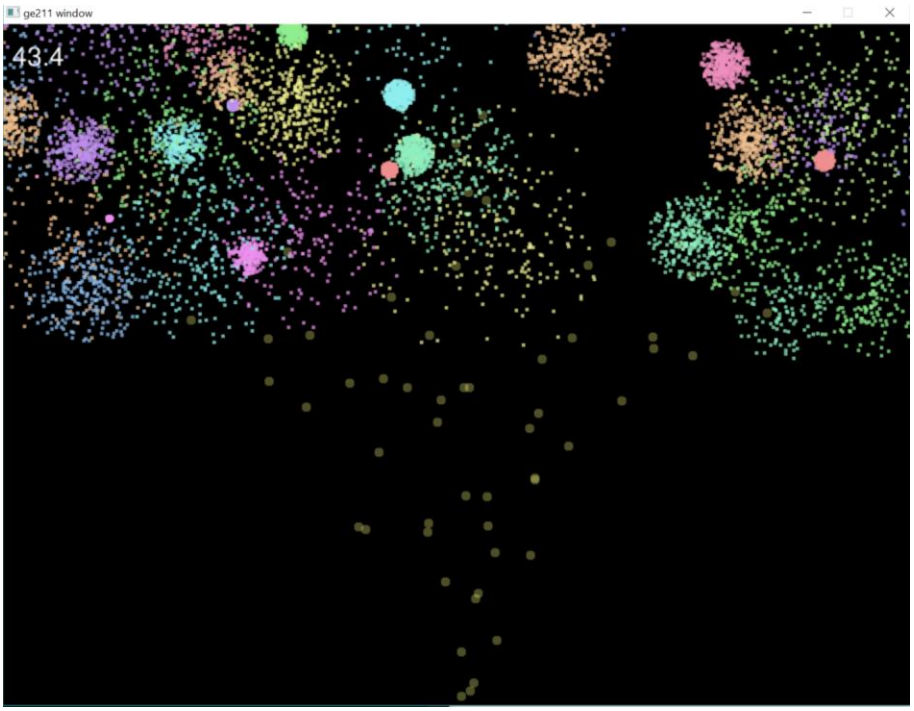

Figure 2: Fireworks in ge211. Framerate in upper left corner

It would also be fruitful to compare the advantages of individual game engines within Rust – what makes one engine preferable to another? What affordances do they provide to a hobbyist, or perhaps even a professional solo developer? These are key questions for further research. In Figure 3, you can see a screenshot from the website http://arewegameyet.com/, which hosts information regarding the state of gaming and game creation using Rust. It aptly describes the situation within the Rust ecosystem: *almost*. Almost is exactly right. Support isn't widespread given the language's recency, and apart from a few fervent enthusiasts and developers, will probably remain that way. Real-world development will continue to occur in Unity or other industry-based game engines, and as it stands, Rust game development will be most applicable to hobbyists or those interested in a new challenge. C++ is just as performant, from the research I have done (although further, more specific insights would be useful), and additionally it has more resources and followers. As it stands, if you are interested in modern game development in Rust, you just have to "… bring your own glue."

Appendix:

presentation link:

https://docs.google.com/presentation/d/1iglYWDjeOalSe_Ow37NOc8Nwkba0e7MiLvHmH5zKjPA/edit?usp=sharing

simple_games_ge211, snake.cpp

```
#include <ge211.h>

#include <vector>
#include <deque>

using namespace ge211;
using namespace std;

// MODEL CONSTANTS

Rectangle const scene_range{0, 0, 1024, 768};
```

```
int const half_sprite_size{5};
int const sprite_size{2 * half_sprite_size};
int const min_x_coord{0};
int const max_x_coord{scene_range.width / sprite_size - 1};
int const min_y_coord{0};
int const max_y_coord{scene_range.height / sprite_size - 1};

//maximum amount of food at once on screen
int const max_food{3};


// MODEL DATA DEFINITIONS

struct Food {
    vector<Position> locs;
};

struct Snake {
    enum class Direction { up, left, down, right };

    Direction dir = Direction::left;
    deque<Position> segments;

    void grow();
    void update();
};

struct Model {
    Food food;
    Snake snake;

    void add_random_food(Random&);
    void add_snake_start(Random&);
    bool food_collision();
    bool self_collision();
    bool out_of_bounds();
    void update();
};

// VIEW DATA DEFINITIONS

struct View {
    Circle_sprite food_sprite{half_sprite_size, Color::medium_magenta()};
    Circle_sprite snake_sprite{half_sprite_size, Color::medium_green()};
    Circle_sprite lose_sprite{half_sprite_size, Color::medium_red()};

    // Maps the virtual position of a sprite to its physical pixel position.
    static Position map_sprite(Position vp);
};

struct Simple_snake : Abstract_game {

    // Model
    Model model;

    // View
    View view;
```

```
    Dimensions initial_window_dimensions() const override;
    void draw(Sprite_set& sprites) override;

    // Controller
    bool game_over{false};
    bool is_paused{false};
    double last_update{0};
    double const reset_update{0.05};
    void on_start() override;
    void on_key(Key key) override;
    void on_frame(double dt) override;
};

// HELPERS

Position random_position(Random&, int min_x, int max_x, int min_y, int
max_y);

// MAIN

int main() {
    Simple_snake{}.run();
}

// FUNCTION DEFINITIONS FOR MODEL

// ~~~MODEL~~~
void Model::add_random_food(Random& rng) {
    while (food.locs.size() < max_food) {
        food.locs.push_back(random_position(rng, min_x_coord, max_x_coord,
                                            min_y_coord, max_y_coord));
    }
}

void Model::add_snake_start(Random& rng) {
    snake.segments.push_back(random_position(rng, min_x_coord, max_x_coord,
                                             min_y_coord, max_y_coord));
}

bool Model::food_collision() {
    Position snake_head = snake.segments.front();
    for (int i = 0; i < food.locs.size(); ++i) {
        if ( food.locs[i] == snake_head ) {
            food.locs.erase(food.locs.begin() + i);
            return true;
        }
    }
    return false;
}

bool Model::self_collision() {
    Position snake_head = snake.segments.front();
    for (int i = 1; i < snake.segments.size(); ++i) {
        if (snake.segments[i] == snake_head) {
            return true;
        }
    }
```

```
    return false;
}

bool Model::out_of_bounds() {
    Position snake_head = snake.segments.front();
    return (snake_head.x < min_x_coord) ||
           (snake_head.x > max_x_coord) ||
           (snake_head.y < min_y_coord) ||
           (snake_head.y > max_y_coord);
}

void Model::update() {
    snake.update();
}

// ~~~SNAKE~~~
void Snake::update() {

    Position new_head = segments.front();
    switch (dir) {
        case Direction::down:
            new_head.y += 1;
            break;
        case Direction::left:
            new_head.x -= 1;
            break;
        case Direction::up:
            new_head.y -= 1;
            break;
        case Direction::right:
            new_head.x += 1;
            break;
    }
    segments.pop_back();
    segments.push_front(new_head);
}

void Snake::grow() {
    Position new_head = segments.back();
    segments.push_back(new_head);
}

// FUNCTION DEFINITIONS FOR VIEW

Position View::map_sprite(Position vp)
{
    return {scene_range.x + sprite_size * vp.x,
            scene_range.y + sprite_size * vp.y};
}

Dimensions Simple_snake::initial_window_dimensions() const {
    return scene_range.dimensions();
}

void Simple_snake::draw(Sprite_set &sprites) {
    for (Position const& loc : model.food.locs) {
        sprites.add_sprite(view.food_sprite, View::map_sprite(loc));
```

```
    }
    Sprite const& segment = game_over? view.lose_sprite : view.snake_sprite;
    for (Position const &pos : model.snake.segments) {
        sprites.add_sprite(segment, View::map_sprite(pos));
    }
}

// FUNCTION DEFINITIONS FOR CONTROLLER

void Simple_snake::on_key(Key key) {

    Snake::Direction cur_dir = model.snake.dir;

    if (key == Key::code('q')) {
        quit();
    } else if (key == Key::code('f')) {
        get_window().set_fullscreen(!get_window().get_fullscreen());
    } else if (key == Key::code('p')) {
        is_paused = !is_paused;
    } else if (key == Key::up() && cur_dir != Snake::Direction::down) {
        model.snake.dir = Snake::Direction::up;
    } else if (key == Key::down() && cur_dir != Snake::Direction::up) {
        model.snake.dir = Snake::Direction::down;
    } else if (key == Key::left() && cur_dir != Snake::Direction::right) {
        model.snake.dir = Snake::Direction::left;
    } else if (key == Key::right() && cur_dir != Snake::Direction::left) {
        model.snake.dir = Snake::Direction::right;
    }
}

void Simple_snake::on_start() {
    model.add_random_food(get_random());
    model.add_snake_start(get_random());
}

void Simple_snake::on_frame(double dt)
{
    if (!is_paused) {
        double time_remaining = last_update - dt;
        if (time_remaining > 0) {
            last_update = time_remaining;
        } else {
            last_update = reset_update + time_remaining;

            if ((!is_paused) && (!game_over))
                model.update();

            if (model.out_of_bounds() || model.self_collision()) {
                game_over = true;
            }

            if (model.food_collision()) {
                model.snake.grow();
                model.add_random_food(get_random());
            }
        }
    }
```

```
}

Position random_position(Random& rng,
                         int min_x, int max_x, int min_y, int max_y)
{
    return {rng.between(min_x, max_x), rng.between(min_y, max_y)};
}
```

simple_games_rust_sdl2, main.rs

```
extern crate sdl2;
extern crate rand;

use sdl2::pixels::Color;
use sdl2::event::Event;
use sdl2::keyboard::Keycode;
//use sdl2::EventPump;
//use sdl2::render::Canvas;

//use std::{thread, time};

// MODEL CONSTANTS

#[derive(Clone, Copy, PartialEq)]
pub enum Direction {
    Right,
    Left,
    Up,
    Down,
}

pub mod simple_snake {
//    use sdl2::rect::Rect;
    use std::collections::VecDeque;
    use super::Direction;
    use rand::{thread_rng, Rng};

    //pub const SCENE_RANGE: Rect = Rect::new(0, 0, 1024, 768);
    pub const SCENE_RANGE_X: u32 = 1024;
    pub const SCENE_RANGE_Y: u32 = 768;
    pub const HALF_SPRITE_SIZE: u32 = 5;
    pub const SPRITE_SIZE: u32 = 2 * HALF_SPRITE_SIZE;
    pub const MIN_X: u32 = 0;
    pub const MAX_X: u32 = SCENE_RANGE_X / SPRITE_SIZE - 1;
    pub const MIN_Y: u32 = 0;
    pub const MAX_Y: u32 = SCENE_RANGE_Y / SPRITE_SIZE - 1;

    pub const MAX_FOOD: usize = 3;

    #[derive(Clone, PartialEq)]
    pub struct Position {
        pub x: u32,
        pub y: u32,
    }

    pub struct Food {
```

```
        pub locs: Vec<Position>,
    }

    impl Food {
        fn new() -> Food {
            let locs: Vec<Position> = vec![];

            Food { locs }
        }
    }

    pub struct Snake {
        pub dir: Direction,
        pub segments: VecDeque<Position>,
    }

    impl Snake {
        fn new() -> Snake {
            let dir: Direction = Direction::Left;
            let segments: VecDeque<Position> = VecDeque::new();

            Snake { dir, segments }
        }
        fn update(&mut self) {
            let mut new_head: Position =
self.segments.front().unwrap().clone();
            match self.dir {
                Direction::Down => new_head.y += 1,
                Direction::Up => new_head.y -= 1,
                Direction::Left => new_head.x -= 1,
                Direction::Right => new_head.x += 1,
            }
            self.segments.pop_back();
            self.segments.push_front(new_head);
        }

        fn grow(&mut self) {
            let new_head: Position = self.segments.back().unwrap().clone();
            self.segments.push_back(new_head);
        }
    }

    pub struct Model {
        pub food: Food,
        pub snake: Snake,
    }

    impl Model {
        fn new() -> Model {
            let food: Food = Food::new();
            let snake: Snake = Snake::new();

            Model { food, snake }
        }

        fn add_random_food(&mut self) {
            while self.food.locs.len() < MAX_FOOD {
```

```
                self.food.locs.push(
                    random_position(MIN_X, MAX_X, MIN_Y, MAX_Y)
                );
            }
        }

        fn add_snake_start(&mut self) {
            self.snake.segments.push_back(
                random_position(MIN_X, MAX_X, MIN_Y, MAX_Y)
            );
        }

        fn food_collision(&mut self) -> bool {
            let snake_head: &Position =
self.snake.segments.front().expect("expected front food_collision");
            for i in 0..self.snake.segments.len() {
                if &self.food.locs[i] == snake_head {
                    self.food.locs.remove(i);
                    return true
                }
            }
            false
        }

        fn self_collision(&self) -> bool {
            let snake_head: &Position =
self.snake.segments.front().expect("expected front self_collision");
            for i in 1..self.snake.segments.len() {
                if &self.snake.segments[i] == snake_head {
                    return true
                }
            }
            false
        }

        fn out_of_bounds(&self) -> bool {
            let snake_head: &Position =
self.snake.segments.front().expect("expected front out_of_bounds");
            snake_head.x < MIN_X || snake_head.x > MAX_X || snake_head.y <
MIN_Y || snake_head.y > MAX_Y
        }

        fn update(&mut self) {
            self.snake.update()
        }
    }

    pub struct SimpleSnake {
        pub model: Model,
        pub game_over: bool,
        pub is_paused: bool,
    }

    impl SimpleSnake {
        pub fn new() -> SimpleSnake {
            let game_over: bool = false;
            let is_paused: bool = false;
```

```
            let model: Model = Model::new();

            SimpleSnake {
                model,
                game_over,
                is_paused,
            }
        }

        pub fn on_start(&mut self) {
            self.model.add_random_food();
            self.model.add_snake_start();
        }

        pub fn update(&mut self) {
            if !self.is_paused && !self.game_over {
                self.model.update();
            }
            if self.model.out_of_bounds() || self.model.self_collision() {
                self.game_over = true;
            }
            if self.model.food_collision() {
                self.model.snake.grow();
                self.model.add_random_food();
            }
        }
    }

    pub fn random_position(min_x: u32, max_x: u32, min_y: u32, max_y: u32) ->
Position {
        let mut rng = thread_rng();
        let x: u32 = rng.gen_range(min_x, max_x);
        let y: u32 = rng.gen_range(min_y, max_y);
        let pos = Position { x, y };
        pos
    }
}


pub fn main() {
    let sdl_context = sdl2::init().unwrap();
    let video_subsystem = sdl_context.video().unwrap();

    let window = video_subsystem.window("simple snake game: sdl2", 1024, 768)
        .position_centered()
        .opengl() // unnecessary?
        .build()
        .unwrap();

    let mut canvas = window.into_canvas().build().unwrap();

    canvas.set_draw_color(Color::RGB(255, 255, 255));
    canvas.clear();
    canvas.present();

    let mut event_pump = sdl_context.event_pump().unwrap();
```

```
    let mut simple_snake = simple_snake::SimpleSnake::new();
    simple_snake.on_start();

    let mut frame: u32 = 0;

    'running: loop {
        for event in event_pump.poll_iter() {
            let cur_dir: Direction = simple_snake.model.snake.dir;

            match event {
                Event::Quit {..} | Event::KeyDown { keycode:
Some(Keycode::Escape), .. } => {
                    break 'running
                },
                Event::KeyDown { keycode: Some(Keycode::Left), .. } => {
                    if cur_dir != Direction::Right {
                        simple_snake.model.snake.dir = Direction::Left;
                    }
                }
                Event::KeyDown { keycode: Some(Keycode::Right), .. } => {
                    if cur_dir != Direction::Left {
                        simple_snake.model.snake.dir = Direction::Right;
                    }
                }
                Event::KeyDown { keycode: Some(Keycode::Up), .. } => {
                    if cur_dir != Direction::Down {
                        simple_snake.model.snake.dir = Direction::Up;
                    }
                }
                Event::KeyDown { keycode: Some(Keycode::Down), .. } => {
                    if cur_dir != Direction::Up {
                        simple_snake.model.snake.dir = Direction::Down;
                    }
                }
                Event::KeyDown { keycode: Some(Keycode::P), .. } => {
                    simple_snake.is_paused = !simple_snake.is_paused;
                }
                _ => {}
            }
        }
        //thread::sleep(time::Duration::from_millis(10));

        if frame >= 30 {
            simple_snake.update();
            frame = 0;
        }

        canvas.set_draw_color(Color::RGB(255, 255, 255));
        canvas.clear();
        canvas.present();

        if !simple_snake.is_paused {
            frame += 1;
        }
    }
}
```

simple_games_ge211, fireworks.cpp (credit, Jesse Tov)

```
#include <ge211.h>

#include <algorithm>
#include <cmath>
#include <iomanip>
#include <vector>

using namespace ge211;
using namespace std;

// MODEL CONSTANTS

Dimensions const scene_dimensions{1024, 768};
Basic_dimensions<double> const gravity_acceleration{0, 120}; // px/s^2
int const min_launch_speed{350}; // px/s
int const max_launch_speed{500}; // px/s
int const max_launch_angle{30}; // degrees from vertical
double const fuse_seconds{2};
int const min_stars{40};
int const max_stars{400};
int const min_star_speed{10}; // px/s
int const max_star_speed{100}; // px/s
double const burn_seconds{2};
int const number_of_colors{12};

// VIEW CONSTANTS

int const mortar_radius = 5;
Color const mortar_color{255, 255, 127, 80};
int const star_radius = 2;

// MODEL DATA DEFINITIONS

struct Projectile
{
    using Position = Basic_position<double>;
    using Velocity = Basic_dimensions<double>;

    Position position;
    Velocity velocity;

    void update(double const dt);

    /// Creates a Projectile with the given Position and a random velocity
    /// within the given speed range and angle range.
    static Projectile random(
            Random&,
            Position,
            double min_speed, double max_speed,
            double min_degrees, double max_degrees);
```

```
};

struct Firework
{
    enum class Stage { mortar, stars, done };

    Stage stage;
    Projectile mortar;
    vector<Projectile> stars;
    int star_color;
    double stage_time;

    void update(double const dt);
    static Firework random(Random&, Projectile::Position);
};

struct Model
{
    vector<Firework> fireworks;

    void update(double const dt);
    void add_random(Random&, Projectile::Position);
};

// VIEW DATA DEFINITIONS

struct View
{
    View();

    Font sans{"sans.ttf", 30};
    Text_sprite fps;
    Circle_sprite mortar{mortar_radius, mortar_color};
    vector<Circle_sprite> stars;
};

// MAIN STRUCT AND FUNCTION

struct Fireworks : Abstract_game
{
    // Model
    Model model;

    // View
    View view;
    Dimensions initial_window_dimensions() const override;
    void draw(Sprite_set& sprites) override;

    // Controller
    bool is_paused = false;
    void on_key(Key key) override;
    void on_mouse_up(Mouse_button button, Position position) override;
    void on_frame(double dt) override;
};

int main()
{
```

```
    Fireworks{}.run();
}

// FUNCTION DEFINITIONS FOR MODEL

void Projectile::update(double const dt)
{
    position += velocity * dt;
    velocity += gravity_acceleration * dt;
}

Projectile
Projectile::random(Random& rng, Position position,
                   double min_speed, double max_speed,
                   double min_degrees, double max_degrees)
{
    double speed = rng.between(min_speed, max_speed);
    double radians = M_PI / 180 * rng.between(min_degrees, max_degrees);
    return {position, {speed * cos(radians), speed * sin(radians)}};
}

void Firework::update(double const dt)
{
    switch (stage) {
        case Stage::mortar:
            if ((stage_time -= dt) <= 0) {
                for (Projectile& star : stars) {
                    star.position = mortar.position;
                    star.velocity += mortar.velocity;
                }
                stage_time = burn_seconds;
                stage = Stage::stars;
            } else {
                mortar.update(dt);
            }
            break;

        case Stage::stars:
            if ((stage_time -= dt) <= 0) {
                stage = Stage::done;
            } else {
                for (Projectile& star : stars) {
                    star.update(dt);
                }
            }
            break;

        case Stage::done:
            break;
    }
}

Firework Firework::random(Random& rng, Projectile::Position p0)
{
    Projectile mortar = Projectile::random(rng, p0,
                                           min_launch_speed,
max_launch_speed,
```

```
                                            -90 - max_launch_angle,
                                            -90 + max_launch_angle);

    vector<Projectile> stars;

    int star_count = rng.between(min_stars, max_stars);
    for (int i = 0; i < star_count; ++i) {
        Projectile star = Projectile::random(rng, {0, 0},
                                             min_star_speed, max_star_speed,
                                             0, 360);
        stars.push_back(star);
    }

    int star_color = rng.up_to(number_of_colors);

    return Firework{Stage::mortar, mortar, stars, star_color, fuse_seconds};
}

void Model::update(double const dt)
{
    for (Firework& firework : fireworks) {
        firework.update(dt);
    }

    size_t i = 0;
    while (i < fireworks.size()) {
        if (fireworks[i].stage == Firework::Stage::done) {
            fireworks[i] = move(fireworks.back());
            fireworks.pop_back();
        } else {
            ++i;
        }
    }
}

void Model::add_random(Random& rng, Projectile::Position position0)
{
    fireworks.push_back(Firework::random(rng, position0));
}

// FUNCTION DEFINITIONS FOR VIEW

View::View()
{
    double hue = 0.0;
    double dhue = 360.0 / number_of_colors;

    for (int i = 0; i < number_of_colors; ++i) {
        stars.emplace_back(star_radius, Color::from_hsla(hue, .75, .75,
.75));
        hue += dhue;
    }
}

Dimensions Fireworks::initial_window_dimensions() const
{
    return scene_dimensions;
```

```
}

void Fireworks::draw(Sprite_set& sprites)
{
    view.fps.reconfigure(Text_sprite::Builder(view.sans)
                                << setprecision(3)
                                << get_frame_rate());
    sprites.add_sprite(view.fps, {10, 10});

    for (Firework const& firework : model.fireworks) {
        switch (firework.stage) {
            case Firework::Stage::mortar:
                sprites.add_sprite(view.mortar,
                                   firework.mortar.position.into<int>());
                break;

            case Firework::Stage::stars:
                for (Projectile const& star : firework.stars) {
                    sprites.add_sprite(view.stars[firework.star_color],
                                       star.position.into<int>());
                }
                break;

            // Shouldn't ever happen:
            case Firework::Stage::done:
                break;
        }
    }
}

// FUNCTION DEFINITIONS FOR CONTROLLER

void Fireworks::on_key(Key key)
{
    if (key == Key::code('q')) {
        quit();
    } else if (key == Key::code('f')) {
        get_window().set_fullscreen(!get_window().get_fullscreen());
    } else if (key == Key::code('p')) {
        is_paused = !is_paused;
    } else if (key == Key::code(' ') && !is_paused) {
        auto dims = get_window().get_dimensions();
        auto initial_position = Position{dims.width / 2, dims.height};
        model.add_random(get_random(), initial_position.into<double>());
    }
}

void Fireworks::on_frame(double dt)
{
    if (!is_paused)
        model.update(dt);
}

void Fireworks::on_mouse_up(Mouse_button, Position position)
{
    if (!is_paused)
        model.add_random(get_random(), position.into<double>());
```

```
}
```