2a. My Create Performance Task is a Tic Tac Toe game written in Rust. Its primary purpose is entertainment. It has two modes: Player vs Player and Player vs Machine. The Player vs Player mode is a traditional Tic Tac Toe game where people take turns placing X's and O's. The Player vs Machine mode pits a human player against an bot that can play both X or O at the user's choosing. The bot uses the Minimax algorithm to generate a map of all possible moves when the application is started, ensuring that a player can never win. Further improvements might make the AI beatable by "messing up" the scoring tree while it is compiling the map, or by decreasing how far it can look ahead.

2b. While developing, I ran into many difficulties. Interfacing with native graphics proved to be difficult. All premade libraries that I found were too complicated, bloated, or inflexible for what I wanted to do. I ended up writing an abstraction over OpenGL and pulling it out into a dependency so that I could abstract over platform specific windowing and have the ability to flexibly target exotic platforms, such as WebAssembly. Another problem I encountered was the layout of ui elements. I was considering designing a primitive layout engine, but I decided that would be overkill for this small project. I instead locked the window to a static size and placed the elements at static locations. This made my application less robust, but allowed a user to interact with my game. In order to implement an iterative development process, I focused gaining base functionality, then fixing what was undesirable. After my game was finished, I turned to people around me and asked for feedback to improve my game, implementing the previously described process on their responses. I used only my own libraries in my official code, though my libraries have dependencies of their own.

2c.

```
//Start Code 1
pub fn update_board_hash(&mut self) {
	let mut index = 0;
	for i in 0..self.board_state.len() {
		index += self.board_state[i] as NodeIndex * (3 as NodeIndex).pow(i as u32);
	}
	self.hash = index
}
//End Code 1
//Start Code 2
let Point { x, y } = position;
let window_height = window.get_height();
let board_width = 3.0 * self.block_width;
let bottom = window_height - self.block_height * 3.0;
let right = board_width;
if *x > 0.0
	&& *x < right
	&& *y < window_height
	&& *y > bottom
```

```
        && self.winner.is_none()
        && self.turn_count < 9
{
        let y_index = ((window_height - y) / self.block_height) as usize;
        let index = (x / self.block_width) as usize + (3 * y_index);

        if self.board_state[index] == 0 {
                self.board_state[index] = self.turn;

                self.update_turn();
                self.update_board_hash();
                self.update_winner();
                self.turn_count += 1;
                self.make_ai_turn();
        }
}
```

//End Code 2

My algorithm is in code 2. This algorithm reacts to a user's mouse in relation to the user's gameboard. It allows the user to make valid moves in the game. The algorithm first uses a modified bounding box test to see if the user clicked the gameboard. This simple algorithm usually tests to see if two axis-aligned rectangles collide with each other, but I modified it to work with points instead. I also added a couple other conditions that prevent the user from making a move if the game is over. Then, I implemented an algorithm that takes the x and y coordinates of the user's mouse and changes them into a single index, as the gameboard is represented by a 1D array. Last, if the cell where the user wants to place their piece is empty, it runs many different algorithms that update the game's state, though one of the most important is updating the board hash. This is Code 1. This algorithm takes a gameboard and serializes it into a single number, allowing the computer bot easily make moves. Overall, Code 2 helps update the board with only valid moves and updates the game state.

2d.

//Start Code

```
struct Button<T> {
    rect: Rect<f32>,
    bg_color: Color,
    text: String,
    text_color: Color,
    event_handler: Box<fn(&mut Self, &mut T)>,
}

impl<T> Button<T> {
    pub fn new(
```

```
        rect: Rect<f32>,
        bg_color: Color,
        text: String,
        text_color: Color,
        event_handler: fn(&mut Self, &mut T),
    ) -> Self {
        Button {
            rect,
            bg_color,
            text,
            text_color,
            event_handler: Box::new(event_handler),
        }
    }

    pub fn render(&self, sprite_renderer: &mut SpriteRenderer) {
        sprite_renderer.enable_quad();
        sprite_renderer.draw_rect(&self.rect, &self.bg_color);
        if self.text.len() > 0 {
            sprite_renderer.enable_text();

            let size = self.rect.height * 0.66;
            let y = self.rect.y + (self.rect.height * 0.25);

            sprite_renderer.draw_text(
                &Point::new(self.rect.x + 20.0, y),
                &self.text,
                size,
                &self.text_color,
            );
        }
    }
}

struct ButtonManager<T> {
    nodes: Vec<Button<T>>,
    free_nodes: Vec<usize>,
}

impl<T> ButtonManager<T> {
    pub fn new() -> Self {
        ButtonManager {
            nodes: Vec::new(),
```

```
        free_nodes: Vec::new(),
    }
}

pub fn handle_event(&mut self, event: &Event, data: &mut T) -> bool {
    match event {
        Event::Click { position, .. } => self
            .nodes
            .iter_mut()
            .find(|button| button.rect.contains(&position))
            .map(|button| {
                (button.event_handler)(button, data);
                true
            })
            .unwrap_or(false),
        _ => false,
    }
}

pub fn render(&self, graphics: &mut Graphics) {
    let sprite_renderer = graphics
        .sprite_renderer
        .as_mut()
        .expect("No Sprite Renderer");
    self.nodes
        .iter()
        .for_each(|button| button.render(sprite_renderer));
}

pub fn add_button(&mut self, button: Button<T>) -> usize {
    if self.free_nodes.len() == 0 {
        if let Some(index) = self.free_nodes.pop() {
            self.nodes[index] = button;
            index
        } else {
            self.nodes.push(button);
            self.nodes.len() - 1
        }
    } else {
        self.nodes.push(button);
        self.nodes.len() - 1
    }
}
```

```
}
//End Code
```

I order to manage the complexity of my program, I abstracted buttons into their own struct and registered them under a “button manager” so I wouldn't have to worry about manually writing collision tests, stylings, and reactions for each one. I developed this abstraction to avoid the undesirable alternative of manually writing each buttons collision test and graphics. A button in my game is a rectangle that reacts to user clicks. However, my game separates graphics rendering and reacting to clicks in separate areas. I might render a rectangle and forget to write the button’s reaction, or I could write the reaction but forget to render the button. There would also be a lot of code duplication with this primitive approach. As a result, I decided that creating a “button manager” as well as a “button” would allow me to write the graphics and reaction cod in one place, as well as decrease code duplication. Abstraction is important in software development as it decreases duplication, speeds development, makes code easier to work with, and in my case, decreases the chance of bugs.

3.

```
// Start src/main.rs
extern crate slash;
extern crate ttt;

use slash::{
    graphics::{
        Color,
        Graphics,
        SpriteRenderer,
    },
    primitives::{
        Point,
        Rect,
    },
    subsystems::{
        Event,
        Window,
    },
    App,
    AppState,
    State,
};
use std::{
    any::Any,
    cell::RefCell,
    rc::Rc,
};
```

```
use ttt::{
    Compilation,
    Compiler,
    Node,
    NodeIndex,
    NodeMap,
    AI,
};

pub struct TTTCompilation {
    pub nodes: NodeMap,
    pub nodes_processed: usize,
    pub winners_processed: usize,
    pub nodes_scored: usize,
    pub board_size: u8,
}

impl TTTCompilation {
    pub fn new() -> TTTCompilation {
        TTTCompilation {
            nodes: NodeMap::new(),
            nodes_processed: 0,
            winners_processed: 0,
            nodes_scored: 0,
            board_size: 3,
        }
    }

    pub fn set_board_size(&mut self, size: u8) {
        self.board_size = size;
    }
}

impl Compilation for TTTCompilation {
    fn inc_nodes_processed(&mut self) {
        self.nodes_processed += 1;
    }

    fn get_nodes_processed(&self) -> usize {
        return self.nodes_processed;
    }

    fn inc_winners_processed(&mut self) {
```

```
    self.winners_processed += 1;
}

fn get_winners_processed(&self) -> usize {
    return self.winners_processed;
}

fn inc_nodes_scored(&mut self) {
    self.nodes_scored += 1;
}

fn get_nodes_scored(&self) -> usize {
    return self.nodes_scored;
}

fn get_node_mut(&mut self, id: NodeIndex) -> &mut Node {
    return self.nodes.get_mut(&id).unwrap();
}

fn insert_node(&mut self, id: NodeIndex, n: Node) {
    self.nodes.insert(id, n);
}

fn contains_node(&self, id: &NodeIndex) -> bool {
    return self.nodes.contains_key(id);
}

fn get_cloned_map(&self) -> NodeMap {
    return self.nodes.clone();
}

fn get_winner(&self, id: &NodeIndex) -> u8 {
    return get_winner(id, self.board_size);
}

fn get_child_states(&self, id: NodeIndex, team: u8) -> Vec<NodeIndex> {
    let mut temp_id = id.clone();
    let mut states = Vec::new();

    for i in 0..(self.board_size * self.board_size) as u32 {
        let num = temp_id % 3;
        if num == 0 {
            let three: NodeIndex = 3;
```

```
                let new_state = id + (team as NodeIndex * three.pow(i));
                states.push(new_state);
            }

            temp_id = temp_id / 3;
        }

        return states;
    }

    fn reset(&mut self) {
        self.nodes = NodeMap::new();
        self.nodes_processed = 0;
        self.winners_processed = 0;
        self.nodes_scored = 0;
        self.board_size = 3;
    }

    fn as_any(&mut self) -> &mut Any {
        self
    }
}

pub fn get_winner(id: &NodeIndex, size: u8) -> u8 {
    let winner = get_winner_row(id, size);
    if winner != 0 {
        return winner;
    }

    let winner = get_winner_col(id, size);
    if winner != 0 {
        return winner;
    }

    let winner = get_winner_diag(id, size);
    if winner != 0 {
        return winner;
    }

    return 0;
}

pub fn get_winner_row(id: &NodeIndex, size: u8) -> u8 {
```

```
    let mut id = id.clone();
    let mut team = 0;

    for _ in 0..size {
        team = id % 3;
        if team == 0 {
            id = id / 27; //3 ^ 3 = 127
            continue;
        }

        for _ in 0..size {
            if team != id % 3 {
                team = 0;
            }

            id = id / 3;
        }

        if team != 0 {
            break;
        }
    }

    return team as u8;
}

pub fn get_winner_col(id: &NodeIndex, size: u8) -> u8 {
    let mut main_id = id.clone();
    let mut team = 0;

    for _ in 0..size {
        let mut id = main_id.clone();
        team = id % 3;

        if team == 0 {
            main_id = main_id / 3;
            continue;
        }

        for _ in 0..size {
            if team != id % 3 {
                team = 0;
            }
```

```
            id = id / 27;
        }

        if team != 0 {
            break;
        }

        main_id = main_id / 3;
    }

    return team as u8;
}

pub fn get_winner_diag(id: &NodeIndex, size: u8) -> u8 {
    let mut main_id = id.clone();
    let mut team = 0;

    for i in 0..2 {
        let mut id = main_id.clone();
        team = id % 3;

        if team == 0 {
            main_id = main_id / 9;
            continue;
        }

        for _ in 0..size {
            if team != id % 3 {
                team = 0;
            }

            id = id / 3u128.pow(4 / (i + 1));
        }

        if team != 0 {
            break;
        }

        main_id = main_id / 9; //3 ^ 2
    }

    return team as u8;
```

```
}
```

```
struct Button<T> {
    rect: Rect<f32>,
    bg_color: Color,
    text: String,
    text_color: Color,
    event_handler: Box<fn(&mut Self, &mut T)>,
}

impl<T> Button<T> {
    pub fn new(
        rect: Rect<f32>,
        bg_color: Color,
        text: String,
        text_color: Color,
        event_handler: fn(&mut Self, &mut T),
    ) -> Self {
        Button {
            rect,
            bg_color,
            text,
            text_color,
            event_handler: Box::new(event_handler),
        }
    }

    pub fn render(&self, sprite_renderer: &mut SpriteRenderer) {
        sprite_renderer.enable_quad();
        sprite_renderer.draw_rect(&self.rect, &self.bg_color);
        if self.text.len() > 0 {
            sprite_renderer.enable_text();

            let size = self.rect.height * 0.66;
            let y = self.rect.y + (self.rect.height * 0.25);

            sprite_renderer.draw_text(
                &Point::new(self.rect.x + 20.0, y),
                &self.text,
                size,
                &self.text_color,
            );
        }
```

```
    }
}

struct ButtonManager<T> {
    nodes: Vec<Button<T>>,
    free_nodes: Vec<usize>,
}

impl<T> ButtonManager<T> {
    pub fn new() -> Self {
        ButtonManager {
            nodes: Vec::new(),
            free_nodes: Vec::new(),
        }
    }

    pub fn handle_event(&mut self, event: &Event, data: &mut T) -> bool {
        match event {
            Event::Click { position, .. } => self
                .nodes
                .iter_mut()
                .find(|button| button.rect.contains(&position))
                .map(|button| {
                    (button.event_handler)(button, data);
                    true
                })
                .unwrap_or(false),
            _ => false,
        }
    }

    pub fn render(&self, graphics: &mut Graphics) {
        let sprite_renderer = graphics
            .sprite_renderer
            .as_mut()
            .expect("No Sprite Renderer");
        self.nodes
            .iter()
            .for_each(|button| button.render(sprite_renderer));
    }

    pub fn add_button(&mut self, button: Button<T>) -> usize {
        if self.free_nodes.len() == 0 {
```

```
            if let Some(index) = self.free_nodes.pop() {
                self.nodes[index] = button;
                index
            } else {
                self.nodes.push(button);
                self.nodes.len() - 1
            }
        } else {
            self.nodes.push(button);
            self.nodes.len() - 1
        }
    }
}
```

```
enum Mode {
    TwoPlayer,
    Computer,
}

struct GameBoard {
    block_width: f32,
    block_height: f32,

    board_state: [u8; 9],
    turn: u8,
    turn_count: u8,
    hash: NodeIndex,
    winner: Option<u8>,
    game_mode: Mode,
    ai_team: u8,

    button_manager: Rc<RefCell<ButtonManager<Self>>>, //It can mutate Gameboard while still existing as a child of it

    ai: AI,

    board_color_1: Color,
    board_color_2: Color,
    background_color: Color,
}

impl GameBoard {
    pub fn update_board_hash(&mut self) {
```

```
    let mut index = 0;
    for i in 0..self.board_state.len() {
        index += self.board_state[i] as NodeIndex * (3 as NodeIndex).pow(i as u32);
    }
    self.hash = index
}

pub fn update_winner(&mut self) {
    let winner = get_winner(&self.hash, 3);
    if winner != 0 {
        self.winner = Some(winner);
    }
}

pub fn restart(&mut self) {
    self.turn = 1;
    self.turn_count = 0;
    self.hash = 0;
    self.winner = None;
    self.board_state = [0; 9];
    self.make_ai_turn();
}

pub fn is_ai_turn(&mut self) -> bool {
    if let Mode::Computer = self.game_mode {
        if self.turn == self.ai_team && self.winner.is_none() && self.turn_count < 9 {
            return true;
        }
    }

    return false;
}

pub fn make_ai_turn(&mut self) {
    if self.is_ai_turn() {
        let hash = self.ai.get_move(self.hash, self.ai_team).expect("oOf");
        self.set_board_from_hash(hash);
        self.update_turn();
        self.hash = hash;
        self.turn_count += 1;
        self.update_winner();
    }
}
```

```
    pub fn set_board_from_hash(&mut self, mut n: NodeIndex) {
        for i in 0..9 {
            if n > 0 {
                self.board_state[i] = (n % 3) as u8;
                n = n / 3;
            } else {
                self.board_state[i] = 0;
            }
        }
    }

    pub fn update_turn(&mut self) {
        self.turn = if self.turn == 1 { 2 } else { 1 };
    }
}

impl State for GameBoard {
    fn new() -> Self {
        let mut compiler = Compiler::new();
        compiler.compilation = Some(Box::new(TTTCompilation::new()));
        compiler
            .init_compilation()
            .expect("Error Starting Compilation");

        while compiler.queue.len() != 0 {
            compiler.process().expect("Error Processing Nodes");
        }

        while compiler.winners.len() != 0 {
            compiler.post_process().expect("Error Scoring Winners");
        }

        while compiler.unscored_nodes.len() != 0 {
            compiler.score_nodes().expect("Error propogating Scores");
        }

        let mut ai = AI::new();
        ai.load(compiler.export().expect("Error Exporting Data"));

        GameBoard {
            block_width: 100.0,
            block_height: 100.0,
```

```
            board_state: [0; 9],
            turn: 1,
            turn_count: 0,
            hash: 0,
            winner: None,
            game_mode: Mode::TwoPlayer,
            ai_team: 1,

            button_manager: Rc::new(RefCell::new(ButtonManager::new())),

            ai,

            board_color_1: Color::from_rgba(255, 0, 0, 255),
            board_color_2: Color::from_rgba(119, 119, 119, 255),
            background_color: Color::from_rgba(48, 48, 48, 255),
        }
    }

    fn init(&mut self, window: &mut Window, graphics: &mut Graphics) {
        let mut button_manager = self.button_manager.borrow_mut();

        let win_width = window.get_width();
        let win_height = window.get_height();

        let border = 10.0;
        let button_width = win_width - (3.0 * self.block_width) - (2.0 * border);

        let bottom = win_height - (self.block_height * 3.0);

        let button_height = 50.0;
        let button_border_vertical = 3.0;

        let mode_button = Button::new(
            Rect::new(
                win_width - button_width - border,
                bottom,
                button_width,
                button_height,
            ),
            Color::from_rgba(0, 0, 0, 255),
            String::from("Two Player"),
            Color::from_rgba(255, 255, 255, 255),
```

```
    |button: &mut Button<Self>, data: &mut Self| {
        if let Mode::TwoPlayer = data.game_mode {
            data.game_mode = Mode::Computer;
            button.text = String::from("Computer");
        } else {
            data.game_mode = Mode::TwoPlayer;
            button.text = String::from("Two Player");
        }
        std::mem::swap(&mut button.bg_color, &mut button.text_color);
        data.restart();
    },
);

let restart_button = Button::new(
    Rect::new(
        win_width - button_width - border,
        bottom + button_height + button_border_vertical,
        button_width,
        button_height,
    ),
    Color::from_rgba(0, 0, 0, 255),
    String::from("Restart"),
    Color::from_rgba(255, 0, 0, 255),
    |_: &mut Button<Self>, data: &mut Self| {
        data.restart();
    },
);

let ai_button = Button::new(
    Rect::new(
        win_width - button_width - border,
        bottom + 2.0 * (button_height + button_border_vertical),
        button_width,
        button_height,
    ),
    Color::from_rgba(0, 0, 0, 255),
    String::from("AI Team: X"),
    Color::from_rgba(255, 0, 0, 255),
    |button: &mut Button<Self>, data: &mut Self| {
        let mut team_str;
        if data.ai_team == 1 {
            data.ai_team = 2;
            button.text_color = Color::from_rgba(0, 0, 255, 255);
```

```
                team_str = "O";
            } else {
                data.ai_team = 1;
                button.text_color = Color::from_rgba(255, 0, 0, 255);
                team_str = "X";
            }

            button.text = format!("AI Team: {}", team_str);
            data.restart();
        },
    );

    button_manager.add_button(mode_button);
    button_manager.add_button(restart_button);
    button_manager.add_button(ai_button);
}

fn handle_event(&mut self, event: &Event, window: &Window) {
    let button_manager_rc = self.button_manager.clone();
    let mut button_manager = button_manager_rc.borrow_mut();
    if button_manager.handle_event(event, self) == true {
        return;
    }

    match event {
        Event::Click { position, .. } => {
            let Point { x, y } = position;
            let window_height = window.get_height();
            let board_width = 3.0 * self.block_width;
            let bottom = window_height - self.block_height * 3.0;
            let right = board_width;

            if *x > 0.0
                && *x < right
                && *y < window_height
                && *y > bottom
                && self.winner.is_none()
                && self.turn_count < 9
            {
                let y_index = ((window_height - y) / self.block_height) as usize;
                let index = (x / self.block_width) as usize + (3 * y_index);

                if self.board_state[index] == 0 {
```

```
                    self.board_state[index] = self.turn;

                    self.update_turn();
                    self.update_board_hash();
                    self.update_winner();
                    self.turn_count += 1;
                    self.make_ai_turn();
                }
            }
        }
        _ => {}
    }
}

fn update(&mut self, _state: &AppState) {}

fn render(&mut self, graphics: &mut Graphics, state: &AppState) {
    graphics.get_error();
    let sprite_renderer = graphics
        .sprite_renderer
        .as_mut()
        .expect("No Sprite Renderer");
    sprite_renderer.enable_quad();
    sprite_renderer.draw_rect(
        &Rect::new(0.0, 0.0, state.width as f32, state.height as f32),
        &self.background_color,
    );

    for i in 0..9 {
        let color = if i % 2 == 0 {
            &self.board_color_1
        } else {
            &self.board_color_2
        };

        let x = (i % 3) as f32 * self.block_height;
        let y = (state.height as f32 - self.block_height) - (i / 3) as f32 * self.block_height;
        sprite_renderer.draw_rect(&Rect::new(x, y, self.block_width, self.block_height), color);
    }

    sprite_renderer.enable_line();
    for i in 0..9 {
        if self.board_state[i] == 1 {
```

```
        let x1 = (i % 3) as f32 * self.block_height;
        let x2 = x1 + self.block_width;
        let y1 = state.height as f32 - (i / 3) as f32 * self.block_height;
        let y2 = y1 - self.block_height;
        sprite_renderer.draw_line(x1, y2, x2, y1, 10.0, &Color::from_rgba(0, 0, 0, 255));
        sprite_renderer.draw_line(x1, y1, x2, y2, 10.0, &Color::from_rgba(0, 0, 0, 255));
    }
}

sprite_renderer.enable_circle();
for i in 0..9 {
    if self.board_state[i] == 2 {
        let radius = self.block_width / 2.0;
        let x = radius + ((i % 3) as f32 * self.block_width);
        let y = state.height as f32 - radius - ((i / 3) as f32 * self.block_height);
        sprite_renderer.draw_circle(x, y, self.block_width, self.block_height);
    }
}

let text_color = Color::from_rgba(255, 255, 255, 255);
sprite_renderer.enable_text();
sprite_renderer.draw_text(
    &Point::new(5.0, 5.0),
    "Welcome to Tic Tac Toe!",
    50.0,
    &text_color,
);

let size = 30.0;
sprite_renderer.draw_text(
    &Point::new(3.0 * self.block_width, state.height as f32 - size),
    &format!("Player Turn: {}", if self.turn == 1 { "X" } else { "O" }),
    size,
    &text_color,
);
sprite_renderer.draw_text(
    &Point::new(3.0 * self.block_width, state.height as f32 - 2.0 * size),
    &format!("Board Hash: {}", self.hash),
    size,
    &text_color,
);
sprite_renderer.draw_text(
    &Point::new(3.0 * self.block_width, state.height as f32 - 3.0 * size),
```

```
            &format!(
                "Winner: {}",
                self.winner
                    .map(|winner| if winner == 1 { "X" } else { "O" })
                    .unwrap_or("None")
            ),
            size,
            &text_color,
        );
        sprite_renderer.draw_text(
            &Point::new(3.0 * self.block_width, state.height as f32 - 4.0 * size),
            &format!("Turn Number: {}", self.turn_count + 1),
            size,
            &text_color,
        );

        self.button_manager.borrow().render(graphics);
    }
}

fn main() {
    let mut app = App::new();
    let mut app_state = AppState::new();

    app_state.width = 480.0;
    app_state.height = 360.0;
    app_state.title = String::from("Tic Tac Toe");

    app.init_app_state(app_state);
    app.set_state(GameBoard::new());
    app.init();

    while app.running && app.main_loop().is_ok() {}
}
//End src/main.rs
//Cargo.toml
[package]
name = "tic_tac_toe"
version = "0.0.1"
edition = "2018"

[dependencies]
```

```
[dependencies.slash]
git = "https://github.com/adumbidiot/slash"
rev = "5c49afa28125a1ac8c0c858fb9251766a061ef25"

[dependencies.ttt]
git = "https://github.com/adumbidiot/TTT"
rev = "0d3628ac897636e5ffb613e1cfcce9dfbc1c9764"
//end Cargo.toml
```