Game Name: Ink Wars
Team: Avi - apal1@wpi.edu, Colin Masucci - cemasucci@wpi.edu
Genre: The genre for this game is a Multiplayer Arcade Action.

Game Description: 2 players battle it out in order to claim as much territory as possible within the time limit. The players claim territory by moving around and leaving a color trail behind them. Each tile claimed adds to the player's score and if a territory is claimed by the opponent will steal the player's points.

Technical Features:
Grid System
Scoreboard (scores and timer)
Player Inputs for 2 players, W A S D and UP DOWN LEFT RIGHT
Ability Actions, Q for player with WASD, CTRL for other player

Artistic Assets:

▨ - Player 1
▩ - Player 2
A cohesive Grid layout:

```
______________________________________________
|□ □ □ □ □ □ □                              ■|
|□ □ □ □ □ □ □                              ■|
|□ □ □ □ □ □ □ □ □ □ □ □ ▨                 ■|
|                                             ■|
|                                             ■|
|                                             ■|
|                                             ■|
|                                   ▩ ■ ■ ■ ■ ■ ■|
|______________________________________________|
| Player1        |    Time Left:    | Player2       |
|_Score:13_____|_______120_______|_Score:25_____|
```

Bomb Ability - covers a 5x5 area around the player
LaserAbility - covers the entire row or column in front of the player, depending on where they look

Audio:
A cool arcade style theme song to play in the background
An explosion sound for the bomb
A futuristic sound for the laser

Implementation Plan:

For Sprites - https://www.asciiart.eu/image-to-ascii,
https://jrgraphix.net/r/Unicode/25A0-25FF

For Sounds - https://freesound.org/, https://pixabay.com/sound-effects/

Distribution of Work:
Colin - implementing player, abilities, color trail, game start & game over, audio

Avi - implementing grid system, border, collision, timer, scoring system

We will both help out when needed and review each other's work. We will both be working together on the promo and presentation towards the end.

Schedule:
Oct. 2 - Plan Listed Above [DONE]

Oct. 6 - Alpha: finish player movement, grid system, color trail, and score [DONE]

Oct. 8 - Final: FIX BUGS (put bin folder in SFML, collision with player and abilities not working), definitely add timer, audio and possibly the 2 abilities [DONE]

Oct. 10 - Promo + Presentation

DESIGN DOCUMENT:

Player Sprites Changed:

Through difficulty dealing with unicode characters (Ex. □▨ ▩■), We ended up switching the characters to numbers. (Player 1: “1” while Player 2: “2”) Then for the trails used (Player 1: “O” while Player 2: “X”).

Attempted Trail Collisions:

We played around with adding interactions for when players collide with the opposing players trail. Similar to how it works in TRON. However after several iterations and some time we were unable to accomplish this.

We believe that part of the problem came from the limitation of the engine as when dealing with small numbers of trail markers, the implementation would usually work however once many markers were introduced into the gameplay, markers started to not spawn properly or bug out in different ways.

An Other version of our player that we discarded can be seen at the bottom of this file.

Grid Implementation:

One factor of dragonfly is that the vertical size of characters are slightly longer than the horizontal length. For our game this meant that grid spaces were not near square visually. To counteract this we overlapped characters slightly to improve this visual, which created grid spaces that were slightly more square without losing grid spaces that the player can travel to.

Sounds:

We picked out sounds to use for our game. Some reused from the initial dragonfly and some were grabbed from other resources.
Our Theme Song (**Crazy Frog - Axel F**)-
https://www.youtube.com/watch?v=k85mRPqvMbE

The Explosion Sound: Reused the explode .wav file from Dragonfly

The Laser Sound and GameOver sound came from Pixaby as .mp3 files -
https://pixabay.com/sound-effects/search/gameover/

Game Layout:

The layout for our game turned out slightly different as we just placed the timer and score towards the top of the screen instead of the bottom. We also added a bunch more art for the end screen to display the winner of the match or if it was a tie meaning both players lost.

Bug Fixes:

We had plenty of bug fixes we figured out. One including the player would sometimes glitch through the walls that we defined. Another was that the player could score points by marking tiles outside of the map. Another was points being incorrectly added up when the player moved over the same tile twice. Another bug was our game incorrectly ending when the timer ran out.

Other Art:

We also played around with many different ASCII art generators using art made on paper then translated into an ASCII representation however we could never get the correct details with the size we were limited to. (seen above)

OLD PLAYER:

```
//Other Player script that is closer to TRON (header)
#pragma once
#include <SFML/Graphics.hpp>
#include "Vector.h"
#include "Object.h"


#include "EventCollision.h"

class Player : public df::Object {

        float PLAYER_SPEED = .5f;
private:
   int m_player_id;              // 1 or 2

private:
   bool m_alive;

public:
   Player(int id, df::Vector start_pos);
   df::Vector m_target_dir;        // current moving direction
        df::Vector m_target_pos;              // current target grid cell
   df::Vector m_prev_pos;             // previous target grid cell

   int eventHandler(const df::Event* p_e) override;
   //int draw() override;

   int handleInput(const df::Event* p_e);
   void update();
   int handleCollision(const df::Event* p_e);


   bool isAlive() const { return m_alive; }
   void updateTarget();

   void leaveMarker();

   df::Vector multiplyVector(const df::Vector& v, float scalar);
```

```
    void die();
    void spawnParticle(const df::Vector& pos);
};
```

```
//Other Player script that is closer to TRON (.cpp)

#include "Player.h"
#include "Vector.h"
#include "Color.h"
#include "Object.h"
#include "Marker.h"
#include "Particle.h"

#include "EventStep.h"
#include "EventKeyboard.h"
#include "EventCollision.h"

#include "WorldManager.h"
#include "DisplayManager.h"



Player::Player(int id, df::Vector start_pos)
    : m_player_id(id), m_alive(true){

    setType("Player");
    if (m_player_id == 1)
        setSprite("player1");
    else
                setSprite("player2");
    setSolidness(df::HARD);
    setPosition(start_pos);
    setAltitude(1);
    // register for events
    registerInterest(df::STEP_EVENT);
        registerInterest(df::KEYBOARD_EVENT);


        //initialize direction and target position
    if (m_player_id == 1) {
        m_target_dir = df::Vector(1, 0);  // start moving right
    }
    else {
        m_target_dir = df::Vector(-1, 0); // start moving left
          }
          updateTarget();
}

int Player::eventHandler(const df::Event* p_e) {
    if (p_e->getType() == df::STEP_EVENT) {
        update();
        return 1;
    }

    if (p_e->getType() == df::KEYBOARD_EVENT) {
                    return handleInput(p_e);
    }

    if (p_e->getType() == df::COLLISION_EVENT) {
        return handleCollision(p_e);
    }
    return 0;
}

int Player::handleInput(const df::Event* p_e) {
    const auto* p_k = dynamic_cast<const df::EventKeyboard*>(p_e);
    if (!p_k)
        return 0;

    if (p_k->getKeyboardAction() == df::KEY_PRESSED) {
        df::Vector dir(0, 0);
        if (m_player_id == 1) {
            // WASD
            if (p_k->getKey() == df::Keyboard::W) dir = df::Vector(0, -1);
            else if (p_k->getKey() == df::Keyboard::S) dir = df::Vector(0, 1);
            else if (p_k->getKey() == df::Keyboard::A) dir = df::Vector(-1, 0);
            else if (p_k->getKey() == df::Keyboard::D) dir = df::Vector(1, 0);
        }
        else {
            // Arrow keys
            if (p_k->getKey() == df::Keyboard::UPARROW) dir = df::Vector(0, -1);
            else if (p_k->getKey() == df::Keyboard::DOWNARROW) dir = df::Vector(0, 1);
            else if (p_k->getKey() == df::Keyboard::LEFTARROW) dir = df::Vector(-1, 0);
            else if (p_k->getKey() == df::Keyboard::RIGHTARROW) dir = df::Vector(1, 0);
        }

        // Ignore zero vector
        if (dir != df::Vector(0, 0)) {

            // Check we are not reversing direction (no 180 degree turns)
            bool reversing_x = (dir.getX() > 0 && m_target_dir.getX() < 0) ||
                (dir.getX() < 0 && m_target_dir.getX() > 0);
            bool reversing_y = (dir.getY() > 0 && m_target_dir.getY() < 0) ||
                (dir.getY() < 0 && m_target_dir.getY() > 0);

            // If the new direction is not opposite of current (so 90 degree turn is allowed)
            if (!(reversing_x || reversing_y)) {

                // Scale Y movement to match X speed
                if (dir.getY() != 0)
                    dir.setY(0.7f * dir.getY());

                m_target_dir = dir;
            }
        }
    }
    return 1;
}

void Player::update() {
    if (!m_alive) return;

          // check if reached target cell/close enough
    if (getPosition().getX()-m_target_pos.getX() < 0.1 && getPosition().getY() -
m_target_pos.getY() < 0.1) {
                    setPosition(m_target_pos);  // snap to target position
        leaveMarker();
        updateTarget();
          }
          df::Vector newDir = multiplyVector(m_target_dir, PLAYER_SPEED); // scale
direction by speed

          setVelocity(newDir);  // set velocity to current target direction
```

```
}

int Player::handleCollision(const df::Event* p_e) {
    const auto* p_c = dynamic_cast<const df::EventCollision*>(p_e);

    // Identify the other object
    df::Object* p_obj1 = p_c->getObject1();
    df::Object* p_obj2 = p_c->getObject2();


    // If colliding with another player
    if (p_obj1->getType() == "Player" || p_obj2->getType() == "Player") {
        die();                          // Current player dies
        return 1;                          // Event handled
    }

    // If colliding with a trail marker
    if (p_obj1->getType() == "Marker" || p_obj2->getType() == "Marker") {
        die();                          // Hit a wall, die
        return 1;
    }

    return 0; // Not handled
}



//int Player::draw() {
//    // simple colored square
//    char c = (m_player_id == 1 ? '1' : '2');
//    df::Color color = (m_player_id == 1 ? df::YELLOW : df::CYAN);
//    DM.drawCh(getPosition(), c, color);
//
//    return 0;
//}

void Player::updateTarget() {
        m_target_pos = getPosition() + m_target_dir;
        m_prev_pos = getPosition();
```

```
}

df::Vector Player::multiplyVector(const df::Vector& v, float scalar) {
   df::Vector result = df::Vector(v.getX() * scalar, v.getY() * scalar);
        return result;
}

void Player::leaveMarker() {
   // Create a marker at the cell we just left
   Marker* m = new Marker(m_player_id, m_prev_pos);
   WM.insertObject(m);
}


void Player::die() {
   m_alive = false;
   setVelocity(df::Vector(0, 0)); // stop moving
   setSolidness(df::SPECTRAL);     // become non-collidable
        //spawnParticle(getPosition());  //causes crash
}

void Player::spawnParticle(const df::Vector& pos) {
   // create a particle (size, lifetime, opacity, color)
   df::Particle* p = new df::Particle(
      0.3f,             // size (cell units)
      30,               // age (lifetime in steps)
      255,              // opacity (0-255)
      df::YELLOW        // color (can also use r,g,b)
   );

   p->setPosition(pos);   // put particle where you want
   WM.insertObject(p);    // register with the world
}
```