COS30031 – Custom Project

Name: Mikkel Aguilar ID: 103613812

**Note – This is not a complete game and only contains game elements I believed can demonstrate what I learned this semester*

Game Overview –

This untitled game is a foundation for a Top-Down shooter, where you take control of a Cowboy to shoot bad guys and collect power ups, cash in on rewards, rinse, and repeat.

For this project, I took inspiration from popular games of the same genre; these include the likes:

Enter The Gungeon – Gameplay, Art Style, Power Ups

SAS Zombie Assault 3 – Gameplay, Shooting

<u>Controls and Interactions</u>

- W, A, S, D to move character and change direction
- Right to shoot with gun, which shoots bullet at the mouse direction
- Walk on power ups to collect them and gain temporary boosts
- Killing enemies will drop coins
- Walk over coins to collect them

Unit Relevant Data Patterns and Techniques Used –

Component Pattern – All game entities (player, enemy, power ups) are built from components, which add functionality and new mechanics to the entity. For example, the animation and buffable components allow entities the

Command Pattern – A command manager that contains all command types controls the movement of an Entity. This is specifically for the single player-controlled entity.

Singleton Pattern – Inside the game class is a singleton, Entity Manager, that has the data for all the entities currently in the game; it is limited to a single instance which allow all entities to gather data from that single instance.

Sound – The SDL mixer was used to play sound effects when certain actions were taken as well as add subtle background music for an enhanced experience

Sprite Cutting – The SDL image library was used to cut sprite sheets to be used for smooth, in game animation

Collision – Square/Rectangle based collision was used to determine when a player (or enemy) walks on a power up and to determine if a bullet hits an entity

Data Structures – Across the entire project, some characters and arrays were never used and instead replaced with other data structures from the standard libraries. Arrays were

instead replaced with vectors for dynamic memory allocation and characters were replaced with strings. Other special data structures used include maps, unique pointers, and pairs.

Game Features and their Implementation –

Animation

Multiple sprite sheets were for each entity, the different directions they can face and the action they are completing. With this, animation was implemented by changing the current position of the source rectangle by the dimension of a single sprite at an interval to create the illusion of constant movement. The program considers if an entity changes action which may result in completely changing the current sprite sheet used.

```
void Animation::update(Entity* entity, SDL_Renderer* r) {
    _frameTime++;

    if (_fps / _frameTime <= _animationPerSecond) {
        _frameTime = 0;
        entity->_rect.x += entity->_frameWidth;
        if (entity->_rect.x >= entity->_textureWidth) {
            entity->_rect.x = 0;
        }
    }

    if (_type == 1) {
        handlePlayerMovementAnimation(entity);
    }
    else if (_type == 2) {
        handleEnemyMovementAnimation(entity);
    }
}
```

Player Movement

The player's movement is controlled by an Input Handler which store the four movement commands that call the entities move functions. The handler checks whether the W, A, S, or D keys have been clicked and adds the respective command to the queue, executing them one by one.

```
void InputHandler::handleInput(Entity* entity) {
    if (entity->keys[SDLK_w]) { commandQueue.push_back(&_wKey); }
    if (entity->keys[SDLK_s]) { commandQueue.push_back(&_sKey); }
    if (entity->keys[SDLK_a]) { commandQueue.push_back(&_aKey); }
    if (entity->keys[SDLK_d]) { commandQueue.push_back(&_dKey); }

    if (!commandQueue.empty()) { executeCommands(entity); }
}

void InputHandler::executeCommands(Entity* entity) {
    while (!commandQueue.empty()) {
        commandQueue.back()->execute(entity);
        commandQueue.pop_back();
    }
}
```

Enemy Wandering

Enemies walk randomly in a direction, and after a time interval, will change the direction it is facing; the enemy always moves the direction they are facing.

```
void RandomMovement::update(Entity* entity) {
    moveTimer += 1;

    if (moveTimer >= 60) {
        int randomType = 0 + ( rand() % ( 3 - 0 + 1 ) );
        if (randomType == 0) {
            entity->_currentDirection = "up";
        }
        else if (randomType == 1) {
            entity->_currentDirection = "down";
        }
        else if (randomType == 2) {
            entity->_currentDirection = "left";
        }
        else if (randomType == 3) {
            entity->_currentDirection = "right";
        }
        moveTimer = 0;
    }
```

Power Ups

Power Ups are spawned in random at locations within the game window, with a random type at an interval using a simple timer. This implementation is possible with the help of the 'random' library that allowed me to get a number in range of the screen, considering the width and height of the power up rectangle.

```
void PowerUp::setRandomLocation() {
    _position.x = 0 + ( rand() % ( (_screenWidth - _frameWidth) - 0 + 1 ) );
    _position.y = 0 + ( rand() % ( (_screenHeight - _frameHeight) - 0 + 1 ) );
}
```

Buffs

When a character picks up a power up, they get a boost specific to the type of power up they collected, these are temporary boosts that change a stat to a pre-set value (no stacking). This functionality is only included for entities with the 'buffable' component; inside this component the list of power ups are iterated over and checked for collision with an entity, a timer then starts and the stat is adjusted. To handle the timing for the boosts, the components checks if a boost is active and slowly decreases the timer until it reaches zero, setting the boost to false and changing to the original stat value.

```
void Buffable::update(Entity* e, SDL_Renderer* r) {
    drawIndicators(e, r);
    handleBoosts(e);
    for (PowerUp* powerUp : *_powerUpList) {
        if (powerUp->collision(e->_position)) {
            if (powerUp->boostType == "damage") {
                Mix_PlayChannel(-1, _damageBoostSound, 0);
                _damageBoosted = true;
                _damageBoostTimer = 300;
                e->_damage = _damage.second;
            }
            else if (powerUp->boostType == "armor") {
                Mix_PlayChannel(-1, _armorBoostSound, 0);
                _armorBoosted = true;
                _armorBoostTimer = 300;
                e->_armor = _armor.second;
            }
            else if (powerUp->boostType == "speed") {
                Mix_PlayChannel(-1, _speedBoostSound, 0);
                _speedBoosted = true;
                _speedBoostTimer = 300;
                e->_speed = _speed.second;
            }
            _powerUpList->erase(remove(_powerUpList->begin(), _powerUpList->end(), powerUp), _powerUpList->end());
        }
    }
}
```

```
void Buffable::handleBoosts(Entity* e) {
    if (_damageBoosted) {
        _damageBoostTimer -= 1;
        if (_damageBoostTimer <= 0) {
            _damageBoosted = false;
            e->_damage = _damage.first;
        }
    }

    if (_armorBoosted) {
        _armorBoostTimer -= 1;
        if (_armorBoostTimer <= 0) {
            _armorBoosted = false;
            e->_armor = _armor.first;
        }
    }

    if (_speedBoosted) {
        _speedBoostTimer -= 1;
        if (_speedBoostTimer <= 0) {
            _speedBoosted = false;
            e->_speed = _speed.first;
        }
    }
}
```

Indicators

If an entity is buffable, indicators will pop up on the screen directly on top of the entity if the entity has the respective Boolean variable for each buff, set to true. The position of each of the indicators are set, but separated between each other so they do not overlap

```
void Buffable::drawIndicators(Entity* e) {
    if (_damageBoosted) {
        _damageIPosition.x = e->_position.x + 16;
        _damageIPosition.y = e->_position.y;
        SDL_RenderCopy(_renderer, _damageIndicator, NULL, &_damageIPosition);
    }
    if (_armorBoosted) {
        _armorIPosition.x = e->_position.x + 31;
        _armorIPosition.y = e->_position.y;
        SDL_RenderCopy(_renderer, _armorIndicator, NULL, &_armorIPosition);
    }
    if (_speedBoosted) {
        _speedIPosition.x = e->_position.x + 46;
        _speedIPosition.y = e->_position.y ;
        SDL_RenderCopy(_renderer, _speedIndicator, NULL, &_speedIPosition);
    }

}
```

Coin System

Coins automatically drop from enemies once their health reaches zero, coins can be picked up by entities with a coin collector component by checking for collision with the entity. The entity manager will increase the coins collected and will remove the coin from the list whilst playing a coin collecting sound

```
if (health <= 0) {
    Coin* coin = new Coin(_renderer, _position.x + 20, _position.y + _position.h / 2, 40, 40, _fps, _screenWidth, _screenHeight);
    EntityManager::get().addToCoinsList(coin);
    EntityManager::get().removeFromEnemiesList(this);
}
```

```
void CoinCollector::update(Entity* e) {
    for (Coin* coin : EntityManager::get().getCoinsList()) {
        if (coin->collision(e->_position)) {
            Mix_PlayChannel(-1, _coinCollectedSound, 0);
            EntityManager::get().coinsCollected += 1;
            EntityManager::get().removeFromCoinsList(coin);
        }
    }
}
```

Health bars

Any entity with a health bar component has a health bar rendered directly below them, which visually represent the health bars in 6 stages. The current texture depends on the threshold (calculated with the maximum health, number of sprites) and current health values which is updated every frame.

```
void HealthBar::update(Entity* e) {
    int threshold = e->startingHealth / 6;

    if (e->health >= e->startingHealth - (threshold * 1)) {
        _currentTexture = _textures[5];
    }
    else if (e->health >= e->startingHealth - (threshold * 2)) {
        _currentTexture = _textures[4];
    }
    else if (e->health >= e->startingHealth - (threshold * 3)) {
        _currentTexture = _textures[3];
    }
    else if (e->health >= e->startingHealth - (threshold * 4)) {
        _currentTexture = _textures[2];
    }
    else if (e->health >= e->startingHealth - (threshold * 5)) {
        _currentTexture = _textures[1];
    }
    else if (e->health >= e->startingHealth - (threshold * 6)) {
        _currentTexture = _textures[0];
    }

    drawBar(e);
}
```

Shooting

An entity with the ranged weapon component can shoot towards the position of the mouse pointer. When a mouse Is clicked the a projectile is added to the projectiles list with the position parameters, and the projectile class itself calculates the bullet trajectory with

triangular math formulas. Shooting also stops characters from moving, creating a delay for the animation to play through.

```
Projectile* projectile;
int x, y;
int x_bullet_pos = entity->_position.x + (entity->_frameWidth);
int y_bullet_pos = entity->_position.y + (entity->_frameHeight);
SDL_GetMouseState(&x, &y);

int x_diff = x - x_bullet_pos;
int y_diff = y - y_bullet_pos;
```

```
if (_type == "gun") {
    projectile = new Projectile(_renderer, x_bul
}
else {
    cout << "Incorrect Projectile Type" << endl;
}
_projectiles.push_back(projectile);
```

```
_angle = atan2(y - target_y, x - target_x);
_xVel = cos(_angle) * _speed;
_yVel = sin(_angle) * _speed;
```