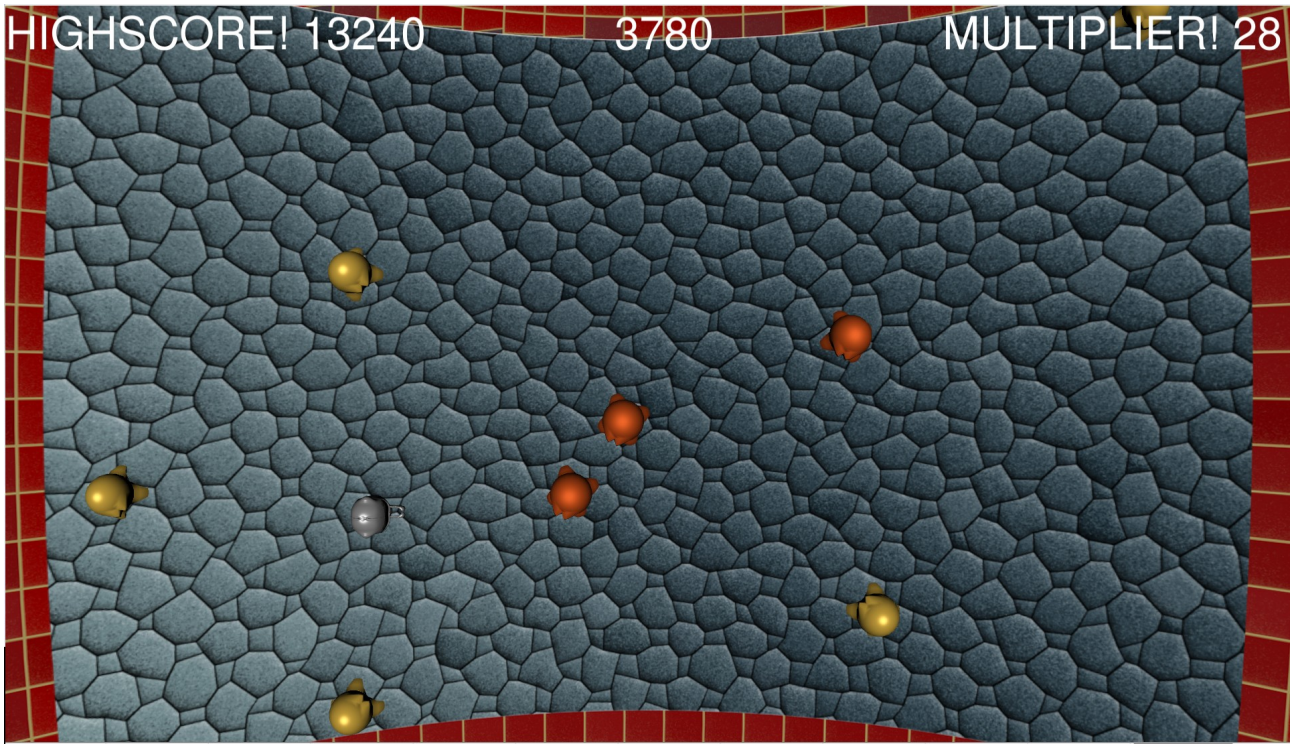
Alex Cowell i7460122 Computing for Animation Documentation



I aimed to create an top down action arcade game similar in gameplay to an old Zelda game. Monsters spawn continuously and assault the player, who has the ability to attack and kill the monsters for points at close range using a sword weapon. The aim is to get a high score by killing as many monsters as possible without dying.

I had greater ambitions with this game, such as obstacles to break open, but I ended up scrapping these ideas as this would have involved implementing a higher-level AI into the enemies so that they did not get stuck behind objects while chasing the player, which I didn't get to try out in the end.

The game works through a gameloop which runs every frame, calling the necessary functions to make everything work. The classes I ended up with were:

- Background
- Enemy
- Heart
- Player
- Sword
- NGLScene (the main one with the gameloop)

Background was simply to create the scenery image as seen in the picture, and need not much explanation. Sword and heart are objects in the game that get displayed every now and then due to certain actions.

Player and enemy are the most complex, particularly enemy, as I used a std::vector to contain every enemy in the scene. This is what it included:

```
/// @brief struct for enemy data
struct enemies
{
  /// @brief position vector
  ngl::Vec3 m_pos;
  /// @brief a value for enemy health
  int health = 1;
  /// @brief a transform stack
  ngl::Transformation m_transform;
  /// @brief the enemy's rotation
  float m_rotation;
  /// @brief the enemy mesh
  std::unique_ptr<ngl::Obj>m_mesh;
};
```

Each enemy had its own position, health value, and rotation and movement, as well as a mesh. They all moved independently of one another, although in the enemy class I also implemented a collision detector to make sure the enemies didn't bunch up. This went through each enemy in the std::vector and compared that enemy with every other one. The method of collision detection was comparing the position of one entity to the selected entity, and moving that selected entity away from the other one if it was close enough, in a direction determined by the comparison statements. I used this method of collision detection for the enemies between themselves as well as the player and enemies- both with them hitting the player, and the player's attack hitting the enemies. I also factored in rotation with the enemy hitting the player, so that the player could then be knocked back in the right direction. Below is the collision code for that knockback scenario, with spos being the player's position and e.m_pos being the enemy's positoin.

```
  // compare the location of the enemy to the player to see if they've been hit and from which direction
  if(spos.m_x - e.m_pos.m_x < 4 && spos.m_x - e.m_pos.m_x >= 0)
  {
    if(spos.m_y - e.m_pos.m_y < 4 && spos.m_y - e.m_pos.m_y >= 0)
    {
      hit = 1;
    }
  }

  if(spos.m_x - e.m_pos.m_x > -4 && spos.m_x - e.m_pos.m_x <= 0)
  {
    if(spos.m_y - e.m_pos.m_y < 4 && spos.m_y - e.m_pos.m_y >= 0)
    {
      hit = 2;
    }
  }

  if(spos.m_x - e.m_pos.m_x < 4 && spos.m_x - e.m_pos.m_x >= 0)
  {
    if(spos.m_y - e.m_pos.m_y > -4 && spos.m_y - e.m_pos.m_y <= 0)
    {
      hit = 3;
    }
  }

  if(spos.m_x - e.m_pos.m_x > -4 && spos.m_x - e.m_pos.m_x <= 0)
  {
    if(spos.m_y - e.m_pos.m_y > -4 && spos.m_y - e.m_pos.m_y <= 0)
    {
      hit = 4;
    }
  }
```

The other main gameplay mechanics are fairly simple to explain. The player controls their movement with either WASD or the arrow keys, but cannot move past the edges of the screen due to a set of if statements that detect this and negate the movement if attempted. The sword is triggered by the attack button (space) which swings around starting at a rotation based on the direction of the attack (the attacks are also directional based on the current movement direction). This is on a cooldown so that it's not possible to spam attacks, which also works as a counter for the swinging animation.

The enemies have the most basic possible AI of moving as directly to the player as possible (only in 8 possible directions, however). They can have varying amounts of health, with it starting out only possible to see enemies who take one hit to kill, but after killing several of these, a chance of seeing enemies who need to be hit twice comes into play, and once more later for the hardest enemy taking 3 hits to kill. Their health is also represented by their colours, which match the player too, and go down each time they're hit- red is one hit, yellow two and grey is three hits. Their speed is also set, but rises by a small amount every time one enemy is killed, making for a gradual difficulty increase throughout the game.

There is also a heart that can spawn if the player is on two or less health (i.e. has been hit at least once) which restores one health to the player. It has a random chance, using std::rand() to spawn every time an enemy is killed, being set at a 10% chance currently.

The game has two files it can read- one is config.txt which can be used to set the initial resolution of the game, affecting the text renders for the HUD, but the window can be resized if necessary while playing. The other file is a score file, which keeps the highest score achieved in the game, and is written to if this is overtaken upon death. Scores also have a multiplication function- each time an enemy is killed, a base 10 points are scored, but then the 'multiplier' goes up by one, and every enemy kill actually gains 10 times the multiplier. The multiplier is reset to one if the player gets hit, so it's imperative to avoid the enemies if a high score is to be achieved.

Conclusion

I'm happy with the result, if not simple in nature, it still achieves what I set out to accomplish. Using std::vectors to spawn multiple enemies of one type at once was satisfying, successful and rounded off my game well. There are many small additions that I would make were I to work on it further, such as a round system where all enemies are killed off and then more spawn, increasing in number, and obstacles as I stated, but I worked to polish what I had achieved into as functional a game and a coding project as possible.