Space Invaders from Scratch - Part 4

Nick Tasios — 2018-05-26 10:51

In this series of posts, I am going to create a clone of the classic arcade game, space invaders (https://en.wikipedia.org/wiki/Space_Invaders), in C++ using only a few dependencies. In this post I will add processing of player input from the keyboard and firing of projectiles.

The complete code of this post can be found here (https://github.com/Grieverheart/space_invaders/blob/caac012ff35bd925e788bb972f41a3a1ad14457a/main.cpp).

Implementing the Key Callback for GLFW

GLFW uses callbacks for passing important events, just like the one we implemented for catching errors. It shouldn't then come as a surprise that we need to implement an appropriate callback function for catching input events. The callback should have the following signature,

```
typedef void(*GLFWkeyfun)(GLFWwindow*, int, int, int, int)
```

and it is set by calling the GLFW function glfwSetKeyCallback (http://www.glfw.org/docs/latest/group__input.html#ga7e496507126f35ea72f01b2e6ef6d155).

Let's first implement a simple key callback for catching the press of the Esc key. When we detect that this key is pressed we will quit the game. This way we can also quickly test our callback. For this, we add a global variable,

```
bool game_running = false;
```

which we set to true before the main loop. Additionally, in the main loop we check if the variable is still true,

```
while (!glfwWindowShouldClose(window) && game_running)
```

if not, the loop is exited and the game is terminated. Finally, we implement the key callback,

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods){
    switch(key){
    case GLFW_KEY_ESCAPE:
        if(action == GLFW_PRESS) game_running = false;
        break;
    default:
        break;
    }
}
```

here mods indicates if any key modifiers are pressed, such as Shift, Ctrl, etc. The scancode is a system-specific code for the key, which we don't use. Finally, when initializing the windows, we set the GLFW callback,

```
glfwSetKeyCallback(window, key_callback);
```

If everything is done correctly, you should be able to quit the game by pressing Esc.

Adding Player Movement

To make things more interesting we're going to add player movement using the left and right arrow keys on the keyboard. We first add a global variable indicating the direction of movement,

```
int move_dir = 0;
```

We do this by assigning a value of +1 for the right arrow, and -1 for the left one. If one of the keys is pressed, its value is added to move_dir, while if it's release, it is subtracted. If e.g. both keys are pressed, move_dir = 0. We implement this logic by adding two additional switch cases to our key callback,

```
case GLFW_KEY_RIGHT:
    if(action == GLFW_PRESS) move_dir += 1;
    else if(action == GLFW_RELEASE) move_dir -= 1;
    break;
case GLFW_KEY_LEFT:
    if(action == GLFW_PRESS) move_dir -= 1;
    else if(action == GLFW_RELEASE) move_dir += 1;
    break;
```

In the main loop, after drawing the game, we add some logic for updating the player position based on the key input,

```
int player_move_dir = 2 * move_dir;

if(player_move_dir != 0)
{
    if(game.player.x + player_sprite.width + player_move_dir >= game.width)
    {
        game.player.x = game.width - player_sprite.width;
    }
    else if((int)game.player.x + player_move_dir <= 0)
    {
        game.player.x = 0;
    }
    else game.player.x += player_move_dir;
}
```

In the inner if statements, we first check for the cases where the player is close to the boundary. In this case we limit the movement of the player sprite so that it remains inside the game boundary.

Adding Projectile Firing

Adding firing of projectiles/bullets, is a bit more involved. Like in the previous section, we add a global variable indicating if the firing button was pressed,

```
bool fire_pressed = 0;
```

and bind the firing to Space by adding one more switch case to the key callback,

```
case GLFW_KEY_SPACE:
    if(action == GLFW_RELEASE) fire_pressed = true;
    break;
```

There are different ways to implement firing of projectiles. I think originally, only one projectile could be present in the game. Here I chose to fire a projectile every time the player releases the button. We then add a new struct for the projectiles, like we did for the Player, Aliens, etc.

```
struct Bullet
{
    size_t x, y;
    int dir;
};
```

where the sign of dir indicates the direction of travel of the projectile, i.e. if it travels upwards, towards the aliens (+), or downwards, towards the player (-). I hope you'll excuse me for referring to projectiles as "Bullets" in the code, but that was what I used originally. We the define the maximum possible projectiles that the game can display, and add them to the Game struct,

```
#define GAME_MAX_BULLETS 128
struct Game
{
    size_t width, height;
    size_t num_aliens;
    size_t num_bullets;
    Alien* aliens;
    Player player;
    Bullet bullets[GAME_MAX_BULLETS];
};
```

At initialization of the game, we set the number of bullets, game.num_bullets = 0, and add a sprite for the projectile,

```
Sprite bullet_sprite;
bullet_sprite.width = 1;
bullet_sprite.height = 3;
bullet_sprite.data = new uint8_t[3]
{
    1, // @
    1, // @
    1  // @
};
```

which, as you can see, we currently draw as a small vertical line. In the main loop, we draw the projectiles similarly to the way we draw the aliens,

```
for(size_t bi = 0; bi < game.num_bullets; ++bi)
{
    const Bullet& bullet = game.bullets[bi];
    const Sprite& sprite = bullet_sprite;
    buffer_draw_sprite(&buffer, sprite, bullet.x, bullet.y, rgb_to_uint32(128, 0, 0));
}
```

After drawing, we update the projectile positions by adding dir, and remove any projectiles that move out of the game area, using a common technique, where we overwrite the element to be deleted with the last one in the array,

```
for(size_t bi = 0; bi < game.num_bullets;)
{
   game.bullets[bi].y += game.bullets[bi].dir;
   if(game.bullets[bi].y >= game.height ||
     game.bullets[bi].y < bullet_sprite.height)
   {
      game.bullets[bi] = game.bullets[game.num_bullets - 1];
      --game.num_bullets;
      continue;
   }

   ++bi;
}
```

Finally, the firing of projectiles by the player is handled at the end of the main loop,

```
if(fire_pressed && game.num_bullets < GAME_MAX_BULLETS)
{
   game.bullets[game.num_bullets].x = game.player.x + player_sprite.width / 2;
   game.bullets[game.num_bullets].y = game.player.y + player_sprite.height;
   game.bullets[game.num_bullets].dir = 2;
   ++game.num_bullets;
}
fire_pressed = false;
```

where we set the projectile direction, dir, to +2

We Need More Aliens for the Invasion

Currently projectiles are non-interactive, they just fly through the swarm of aliens and into the abyss! If we want to stop the alien invasion, we better make the projectiles less wimpy. But before that, let's first add more types of aliens. There are 3 alien types in Space Invaders, in addition, we will implement alien death by adding one more alien type, the dead alien. We implement this in our code using an enum,

```
enum AlienType: uint8_t
{
   ALIEN_DEAD   = 0,
   ALIEN_TYPE_A = 1,
   ALIEN_TYPE_B = 2,
   ALIEN_TYPE_C = 3
};
```

and add a flag in the Alien struct indicating the alien type. We then store the alien sprites as an array indexed by the alien type and animation frame index.

```
Sprite alien_sprites[6];

alien_sprites[0].width = 8;
alien_sprites[0].height = 8;
alien_sprites[0].data = new uint8_t[64]
{
    0,0,0,1,1,0,0,0, // ...@@...
    0,0,1,1,1,1,0,0, // ..@@@@..
    0,1,1,1,1,1,1,0, // .@@@@@@.
    1,1,0,1,1,0,1,1, // @@.@@.@@
    1,1,1,1,1,1,1,1, // @@@@@@@@
    0,1,0,1,1,0,1,0, // .@.@@.@.
    1,0,0,0,0,0,0,1, // @......@
    0,1,0,0,0,0,1,0  // .@....@.
};

alien_sprites[1].width = 8;
alien_sprites[1].height = 8;
alien_sprites[1].data = new uint8_t[64]
{
    0,0,0,1,1,0,0,0, // ...@@...
    0,0,1,1,1,1,0,0, // ..@@@@..
    0,1,1,1,1,1,1,0, // .@@@@@@.
    1,1,0,1,1,0,1,1, // @@.@@.@@
    1,1,1,1,1,1,1,1, // @@@@@@@@
    0,0,1,0,0,1,0,0, // ..@..@..
    0,1,0,1,1,0,1,0, // .@.@@.@.
    1,0,1,0,0,1,0,1  // @.@..@.@
};

alien_sprites[2].width = 11;
alien_sprites[2].height = 8;
alien_sprites[2].data = new uint8_t[88]
{
    0,0,1,0,0,0,0,0,1,0,0, // ..@.....@..
    0,0,0,1,0,0,0,1,0,0,0, // ...@...@...
    0,0,1,1,1,1,1,1,1,0,0, // ..@@@@@@@..
    0,1,1,0,1,1,1,0,1,1,0, // .@@.@@@.@@.
    1,1,1,1,1,1,1,1,1,1,1, // @@@@@@@@@@@
    1,0,1,1,1,1,1,1,1,0,1, // @.@@@@@@@.@
    1,0,1,0,0,0,0,0,1,0,1, // @.@.....@.@
    0,0,0,1,1,0,1,1,0,0,0  // ...@@.@@...
};

alien_sprites[3].width = 11;
alien_sprites[3].height = 8;
alien_sprites[3].data = new uint8_t[88]
{
    0,0,1,0,0,0,0,0,1,0,0, // ..@.....@..
    1,0,0,1,0,0,0,1,0,0,1, // @..@...@..@
    1,0,1,1,1,1,1,1,1,0,1, // @.@@@@@@@.@
    1,1,1,0,1,1,1,0,1,1,1, // @@@.@@@.@@@
    1,1,1,1,1,1,1,1,1,1,1, // @@@@@@@@@@@
    0,1,1,1,1,1,1,1,1,1,0, // .@@@@@@@@@.
    0,0,1,0,0,0,0,0,1,0,0, // ..@.....@..
    0,1,0,0,0,0,0,0,0,1,0  // .@.......@.
};

alien_sprites[4].width = 12;
alien_sprites[4].height = 8;
alien_sprites[4].data = new uint8_t[96]
{
    0,0,0,0,1,1,1,1,0,0,0,0, // ....@@@@....
    0,1,1,1,1,1,1,1,1,1,1,0, // .@@@@@@@@@@.
    1,1,1,1,1,1,1,1,1,1,1,1, // @@@@@@@@@@@@
    1,1,1,0,0,1,1,0,0,1,1,1, // @@@..@@..@@@
    1,1,1,1,1,1,1,1,1,1,1,1, // @@@@@@@@@@@@
    0,0,0,1,1,0,0,1,1,0,0,0, // ...@@..@@...
    0,0,1,1,0,1,1,0,1,1,0,0, // ..@@.@@.@@..
    1,1,0,0,0,0,0,0,0,0,1,1  // @@........@@
};


alien_sprites[5].width = 12;
alien_sprites[5].height = 8;
alien_sprites[5].data = new uint8_t[96]
{
    0,0,0,0,1,1,1,1,0,0,0,0, // ....@@@@....
    0,1,1,1,1,1,1,1,1,1,1,0, // .@@@@@@@@@@.
    1,1,1,1,1,1,1,1,1,1,1,1, // @@@@@@@@@@@@
    1,1,1,0,0,1,1,0,0,1,1,1, // @@@..@@..@@@
    1,1,1,1,1,1,1,1,1,1,1,1, // @@@@@@@@@@@@
    0,0,1,1,1,0,0,1,1,1,0,0, // ..@@@..@@@..
    0,1,1,0,0,1,1,0,0,1,1,0, // .@@..@@..@@.
    0,0,1,1,0,0,0,0,1,1,0,0  // ..@@....@@..
};

Sprite alien_death_sprite;
alien_death_sprite.width = 13;
alien_death_sprite.height = 7;
alien_death_sprite.data = new uint8_t[91]
{
    0,1,0,0,1,0,0,0,1,0,0,1,0, // .@..@...@..@.
    0,0,1,0,0,1,0,1,0,0,1,0,0, // ..@..@.@..@..
    0,0,0,1,0,0,0,0,0,1,0,0,0, // ...@.....@...
    1,1,0,0,0,0,0,0,0,0,0,1,1, // @@.........@@
    0,0,0,1,0,0,0,0,0,1,0,0,0, // ...@.....@...
    0,0,1,0,0,1,0,1,0,0,1,0,0, // ..@..@.@..@..
    0,1,0,0,1,0,0,0,1,0,0,1,0  // .@..@...@..@.
};
```

Note that we also added a death sprite. To keep track of alien deaths, we create an array of death counters,

```
uint8_t* death_counters = new uint8_t[game.num_aliens];
for(size_t i = 0; i < game.num_aliens; ++i)
{
    death_counters[i] = 10;
}
```

at each frame, if an alien is dead, we decrement the death counter, and "remove" the alien from the game when the counter reaches 0. When drawing the aliens, we now need to check if the death counter is bigger than 0, otherwise we don't have to draw the alien. This way, the death sprite is shown for 10 frames.

```
for(size_t ai = 0; ai < game.num_aliens; ++ai)
{
    if(!death_counters[ai]) continue;

    const Alien& alien = game.aliens[ai];
    if(alien.type == ALIEN_DEAD)
    {
        buffer_draw_sprite(&buffer, alien_death_sprite, alien.x, alien.y, rgb_to_uint32(128, 0, 0));
    }
    else
    {
        const SpriteAnimation& animation = alien_animation[alien.type - 1];
        size_t current_frame = animation.time / animation.frame_duration;
        const Sprite& sprite = *animation.frames[current_frame];
        buffer_draw_sprite(&buffer, sprite, alien.x, alien.y, rgb_to_uint32(128, 0, 0));
    }
}
```

Note that we have further updated the drawing loop to draw the appropriate alien sprite based on the type of alien. The death counters are update in a loop before just updating the projectiles,

```
for(size_t ai = 0; ai < game.num_aliens; ++ai)
{
    const Alien& alien = game.aliens[ai];
    if(alien.type == ALIEN_DEAD && death_counters[ai])
    {
        --death_counters[ai];
    }
}
```

Cool Guys Don't Look At Explosions (https://www.youtube.com/watch?v=Sqz5dbs5zmo)

Everything is finally set in place to implement the most satisfying element in Space Invaders; blowing up the aliens! To implement this, we have to check if a bullet hits an alien which is alive, when updating its position,

```
for(size_t ai = 0; ai < game.num_aliens; ++ai)
{
    const Alien& alien = game.aliens[ai];
    if(alien.type == ALIEN_DEAD) continue;

    const SpriteAnimation& animation = alien_animation[alien.type - 1];
    size_t current_frame = animation.time / animation.frame_duration;
    const Sprite& alien_sprite = *animation.frames[current_frame];
    bool overlap = sprite_overlap_check(
        bullet_sprite, game.bullets[bi].x, game.bullets[bi].y,
        alien_sprite, alien.x, alien.y
    );
    if(overlap)
    {
        game.aliens[ai].type = ALIEN_DEAD;
        // NOTE: Hack to recenter death sprite
        game.aliens[ai].x -= (alien_death_sprite.width - alien_sprite.width)/2;
        game.bullets[bi] = game.bullets[game.num_bullets - 1];
        --game.num_bullets;
        continue;
    }
}
```

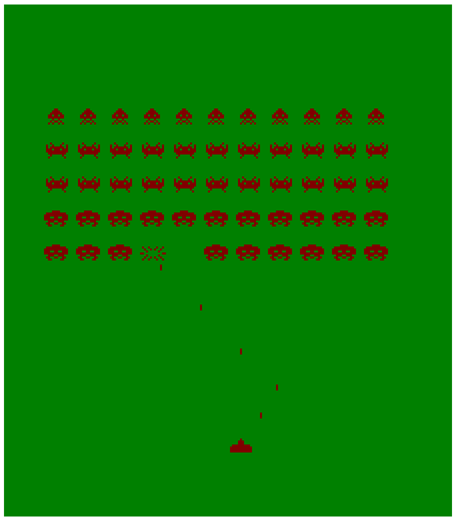
We loop over all aliens and use the function sprite_overlap_check to check if two sprites overlap. We do this by simply checking if the sprite rectangles overlap,

```
bool sprite_overlap_check(
    const Sprite& sp_a, size_t x_a, size_t y_a,
    const Sprite& sp_b, size_t x_b, size_t y_b
)
{
    if(x_a < x_b + sp_b.width && x_a + sp_a.width > x_b &&
       y_a < y_b + sp_b.height && y_a + sp_a.height > y_b)
    {
        return true;
    }

    return false;
}
```

If the bullet sprite overlaps with the alien sprite, we remove the bullet from the game, and change the alien type to ALIEN_DEAD. Note that we use a haphazard way of re-centering the death sprite as we are not properly handling sprite centering in our game.

If you compile this post's code, you should be able to blast some aliens!



Conclusion

In this post, setup the most important part of what makes a game, a game; interactivity. Using GLFW, we bound keys for moving the player and firing projectiles. Even though though the game is still at a very incomplete state, we can already have some fun firing bullets at the aliens. Enjoying the process is very important in game development and is what sparks experimentation of new ideas. You see that we are able to setup a game prototype in a low level programming language with very little effort. More importantly, we can reuse this code to create prototypes for other 2D games.

In the next post we will create the necessary tools to draw basic text on screen, and we will keep track and draw the player's score.