⬅ Project 2 Objects

What you'll be implementing

Click here for the reference manual for this mini-engine.

Please do things in order. Do not skip steps. **A program with ten broken features is not as good as a program with three working features.**

The Player ¶

The **player** will be able to rotate, accelerate, slow down, and fire bullets. They can also be damaged or destroyed by the rocks. When damaged, they will become temporarily invulnerable. When destroyed, they will "respawn" or reappear after a short delay... unless they've run out of retries, in which case it's a game over!

1. **Open `player.asm` and have a look at the code that's there.** Some functions have already been defined for you:
   - `player_init` is called at the very beginning of a game.
     - Look how it sets the `player` object's fields.
     - It also calls...

- `player_respawn`, which resets many (but not all) of the player-related global variables and `player` fields.
- `player_draw` is the implementation of a **method.**
  - This method is called by `Object_draw_all()`, and is passed the `player` object as the `a0` argument.
  - It does some funny stuff to figure out whether to draw the player and what frame of the animation to draw, but...
  - The important bit is the call to `Object_blit_5x5_trans` at the end. It passes the player object as `a0` and the address of an image pattern as `a1`.
- `player_collide_all` loops over all active objects in the `objects` array.
  - if they have a collision method...
  - and they're overlapping the player...
    - then it calls their collision method.

2. **Now to make something happen.** `player_check_input` is the place to start.
   - You should be familiar with checking the input by now, right? ;)
   - First, **call `player_check_input` from the `player_update` method.**

   - Here's the pseudocode for what you should put in `player_check_input`:

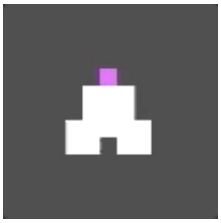
```
v0 = input_get_keys();

// rotate left (counterclockwise)
if((v0 & KEY_L) != 0) {
    // player_angle is a global variable in gl
    player_angle -= PLAYER_ANG_VEL;
    if(player_angle < 0) player_angle += 360;
}

// rotate right (clockwise)
```

```
if((v0 & KEY_R) != 0) {
    player_angle += PLAYER_ANG_VEL;
    if(player_angle >= 360) player_angle -= 36
}
```

If you assemble and run, you should now be able to use the left and right keys to rotate the player, like this! (The purple dot is the front of the ship.)



If it doesn't work...

- If it doesn't turn at all, **be sure you stored the changed angle back into `player_angle`!**
- If it crashes in `display_2204_0207.asm`, it's probably because you **didn't correctly limit the angle.**
  - See the `if(player_angle < 0)` in the pseudocode? That's important.
  - If the angle goes negative or above 359, the code in `player_draw` will malfunction.
- Some other problem? **ASK FOR HELP.**
  - This is just the beginning. Do not move on until this is working.

A debugging tangent

1. Open `hud.asm`. Here you can see the code for drawing the numbers and stuff that appear on the edges of the screen.
2. Comment out the `jal debug_draw_frame_counter` line.
3. Make a new function in there, `debug_draw_angle`. Call it from `hud_draw` like the other debug functions.
4. Have that function display the player's angle using `display_draw_int`. Follow the pattern of the other functions.

Now you should have a display of the player's angle on-screen. This kind of thing can be **immensely** helpful when debugging things.

Feel free to modify the HUD to display any kind of info you want. It's also a good place to put prints to the console using syscalls, if the screen is too limited.

Making the player move

1. Change the contents of `player_update` to the equivalent of this code:

```
// from last step
player_check_input();

// new code!
player_update_thrust();

// be sure to pass the player's address as a0 (la a
// the ALL_CAPS values are constants from constants
// do you use to put a constant in a register?
Object_damp_velocity(player, PLAYER_DRAG);
Object_accumulate_velocity(player);
Object_wrap_position(player);
```

2. In `player_check_input`, add some code to this effect:

```
// player_accel is a global variable, again. I GAVE
if((v0 & KEY_U) != 0)
    player_accel = 1;
else
    player_accel = 0;
```

   - Maybe you could draw the value of `player_accel` on the HUD, to make sure this code works? ;)
3. In `player_update_thrust` ...

```
if(player_accel) {
    Object_apply_acceleration(player, 0, -PLAYER_TH
}
```

4. Test it out.
    - Uh oh. The ship just flies straight up, no matter which way it faces.
    - Time for a tangent.

Velocity and acceleration

If you remember from calculus 1, **position** is the integral of **velocity;** and **velocity** is the integral of **acceleration.**

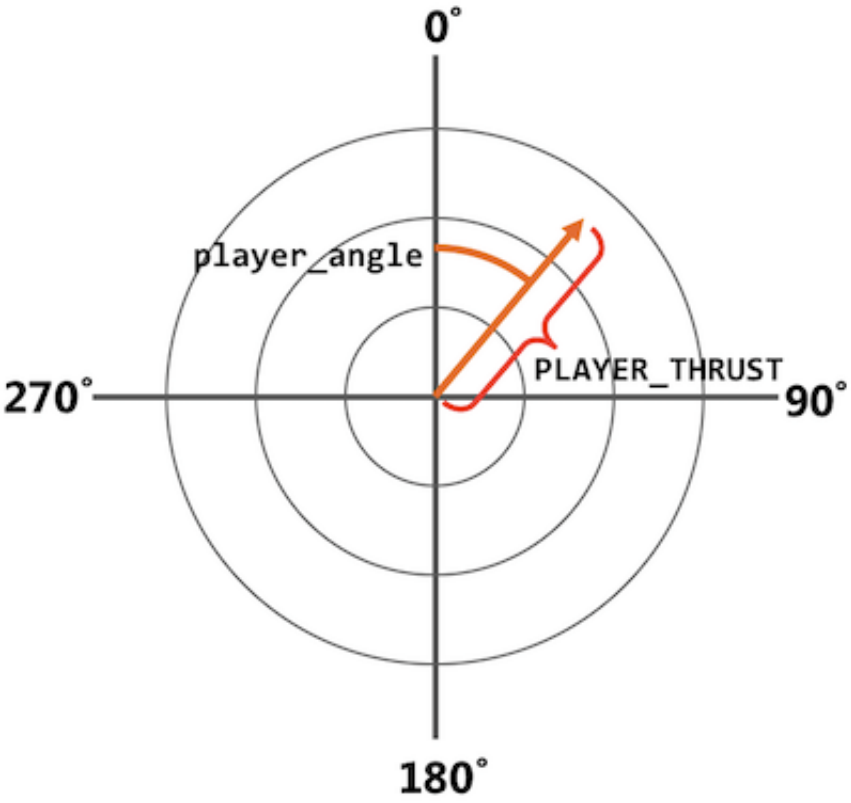
Right now, you are applying an acceleration (with `Object_apply_acceleration`) in a **fixed direction.** That means the object will only ever move up.

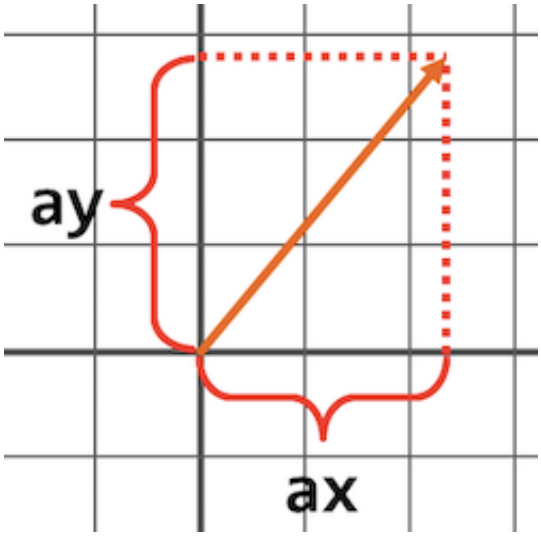
To make the ship move in *any* direction, we need to calculate an acceleration vector that faces in the **same direction the ship is facing.** We have the `player_angle`, and we want the acceleration vector to be `PLAYER_THRUST` long. This sounds like… **polar coordinates?**

The player's **angle** and **thrust** define a point in a polar coordinate system, where 0 degrees is up and 90 degrees is right (flipped from how mathematicians usually define it). See the diagram to the left.

`Object_apply_acceleration` expects the x and y acceleration components as **cartesian coordinates,** like on the right. So how do we go from one to another?

Well, **I wrote a function for you.** Have a look at how it works in `math.asm` if you're curious.





So here's what you have to do in **`player_update_thrust`**:

```
if(player_accel) {
        // this function returns TWO values. thankfully
        v0, v1 = to_cartesian(PLAYER_THRUST, player_angl
```

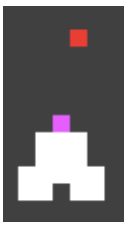
```
        // use 'move'...
        Object_apply_acceleration(player, v0, v1);
}
```

Now you should be able to fly around anywhere you want!

> ***Try this:*** If you're wondering what the stuff in `player_update` is *doing,* one way to find out is to comment it out and see what happens.
>
> Try commenting out `Object_damp_velocity` and hold up. That's fun! But kind of weird, too.
>
> Try commenting out `Object_wrap_position` and fly off the screen. That's less fun, but it's enlightening.
>
> If you comment out `Object_accumulate_velocity`, then it stops moving altogether. The least fun :(

Now let's take a detour to **bullets.**

The Bullets



The **bullets** are fired by the player with the B key. When fired, they travel in the direction the player is facing. They only travel a short time before disappearing. They can destroy rocks.

They are pretty simple objects: they are created, they move, and then they disappear. Collision with rocks is handled later, by the rocks.

1. In `player.asm`:
    - Add this to `player_check_input`:

```
if((v0 & KEY_B) != 0)
    player_fire();
```

- In **`player_fire`**:

```
// remember how to get fields from a struct?
bullet_new(player.x, player.y, player_angle);
```

2. In **`bullet.asm`**, fill in the functions like so:

```
void bullet_new(x, y, angle) {
    // calling another function... what do you have
    // values in the a registers before you do this
    obj = Object_new(TYPE_BULLET);

    // Object_new returns null (0) if it couldn't a
    if(obj != 0) {
        set obj.x and obj.y to the arguments
    }
}

void bullet_update(bullet) {
    Object_accumulate_velocity(bullet);
    Object_wrap_position(bullet);
}

void bullet_draw(bullet) {
    display_set_pixel(bullet.x >> 8, bullet.y >> 8,
}
```

3. Okay! Cool! Let's test it! **Fly around and hit B. What happens?**
   - You should get red bullets appearing wherever the ship is... but **they sit still.**

    - And eventually, you can't shoot any more. (You hit the limit of the number of objects.)
4. First, let's make them move.
    - In `bullet_new`, after setting its position...
    - call `to_cartesian(BULLET_THRUST, angle)` and set the result as the object's **velocity.**
        - (`angle` is the argument to `bullet_new`)
    - **Now they should move.** Really fast. Forever. And your ship sprays them out like water. FUN!

Bullet lifetime

You can think of bullets as a "subclass" of Object, because they have an extra field: `Bullet_frame`. This field is used to implement a **frame timer.** Think of it like a countdown timer, or a ticking time bomb.

- When the bullet is created, you put some value into it (like `10`, or the constant `BULLET_LIFE`).
- Then, **every frame** (in `bullet_update`), you decrement it.
- And after decrementing it, when it reaches 0, **time's up!** The bullet should disappear.

This "frame timer" concept will come up over and over, so practice with it here and make sure you understand it well.

1. In `bullet_new`, set the object's `Bullet_frame` field to `BULLET_LIFE`. Assuming the object is in `s0`:

```
li t0, BULLET_LIFE
sw t0, Bullet_frame(s0)
```

2. Change `bullet_update`'s code to work like this:

```
bullet.Bullet_frame--;
```

```
if(bullet.Bullet_frame == 0)
    Object_delete(bullet);
else {
    Object_accumulate_velocity(bullet);
    Object_wrap_position(bullet);
}
```

3. Now the bullets should **appear, move, and disappear.** Magic.

Of course, the player is still SPRAYING them out like a hose. That's cool, but not right.

By the way, you're done with the bullet object. Good job!

Limiting the firing rate

Finally, let's return to **`player.asm`** to fix that problem. The **`player_fire_time`** variable is a **frame timer** used to limit how quickly they can fire bullets.

1. In **`player_update`**, add code like this before all the other stuff:

```
if(player_fire_time > 0)
    player_fire_time--;
```

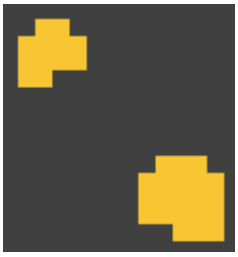
2. In **`player_fire`**, change it to this:

```
if(player_fire_time == 0) {
    player_fire_time = PLAYER_FIRE_DELAY;
    bullet_new(player.x, player.y, player_angle);
}
```

3. And that's it. When you test it, you should only be able to fire about 3 bullets per second.
    - That's part of the challenge!!

Checkpoint: You now have about a 50%. You're halfway there!

The Rocks



The **rocks** are the targets and hazards. There are three sizes: large, medium, and small. The game starts with several large rocks. When destroyed, they create two medium rocks; and when medium rocks are destroyed, they create two small rocks. When all rocks are destroyed, the player wins the game.

The rocks are definitely the most complicated objects, but they have some very easy parts too. Once you finish them, you'll have a 90%. Yeah!

From now on, I will describe things more abstractly, and you will implement them yourself. **It is important for you to learn to do things in a sensible order, and to test them at every step of the way.** Make use of the debugging tools available to you. Get help if you are stuck. You can do this!

First work on making just the large rocks.

1. `rock_new(x, y, type)`

    - This will work a lot like `bullet_new`.
    - This should create a new object of type `type` and set its position to `x` and `y`.
    - It's also important to set its **bounding box size.**
        - Set its `Object_hw` and `Object_hh` to `ROCK_L_HW` and `ROCK_L_HH`, respectively.
    - Finally, set its velocity like `bullet_new`, except...
        - use `random(360)` to pick a random angle (this is a function in `math.asm`)
        - use `ROCK_VEL` as the other argument to `to_cartesian`
2. `rock_update(rock)`
    - Accumulate the velocity
    - Wrap the position
    - And call `rock_collide_with_bullets(rock)`.
3. `rock_draw_l(rock)`
    - `la a1, spr_rock_l`
    - then call `Object_blit_5x5_trans`

Nothing shows up yet. But that's cause `rock_new` was not yet called. That's the job of `rocks_init`. `rocks_init` is called from `main` for you.

First, just to test:

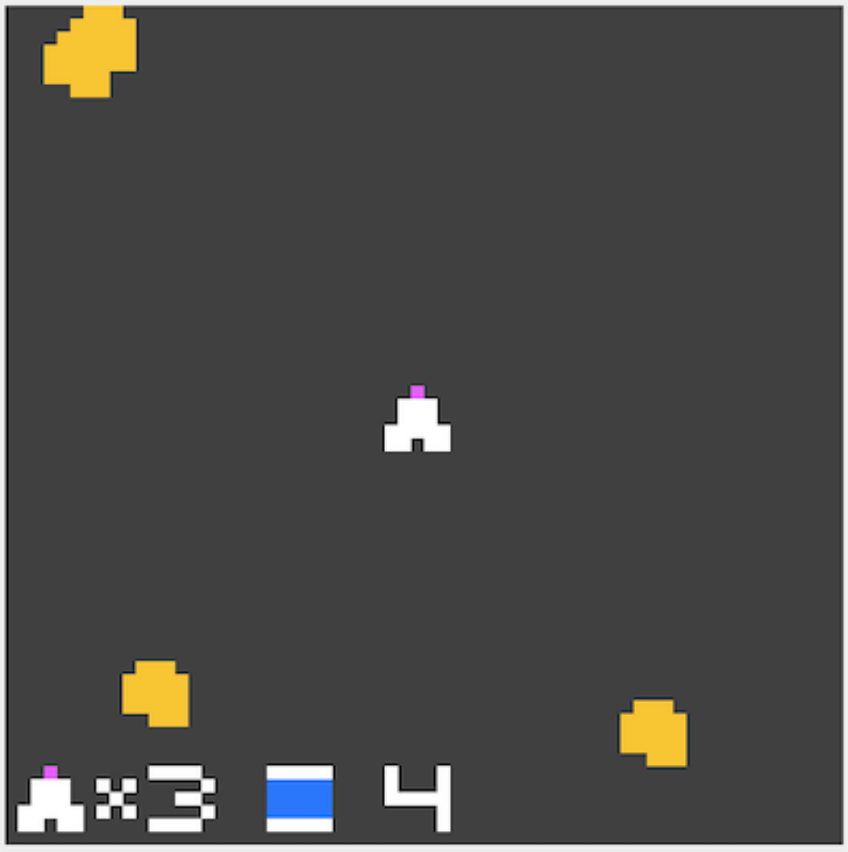
- Inside `rocks_init`, call `rock_new(0x100, 0x100, TYPE_ROCK_L)`.
- One rock should now appear at the top left of the screen!
- Now test it a few times. **Make sure the rock is moving, and that its direction is random every time you run the program.**

Now, **remove that test code from `rocks_init`,** and in its place:

- For i = 0 to *the argument passed to* `rocks_init`:
    - Generate a random x coordinate using this formula: `(random(0x2000) + 0x3000) % 0x4000`
    - Generate a random y coordinate using that formula **a second time**

(don't reuse the same value!)
- Call `rock_new` with those coordinates and `TYPE_ROCK_L`

The rocks should appear all around the edges of the screen! And like magic, that number at the bottom should show how many there are :O



Rock collision

Believe it or not, the player is **already colliding with the rocks!**

Inside `rock_collide_l`, put something like `println_str "BOOM!"`. Then drive your ship into the rocks. You'll see that message get printed to the console whenever your ship touches a rock.

Now change the code inside `rock_collide_l` to just `jal rock_get_hit`, and inside `rock_get_hit`, put `jal Object_delete`. Now your ship erases rocks when it touches them, and the count at the bottom of the screen decreases to 0. Neat!

Note: if it's behaving strangely now, you may have issues with your rocks' bounding box sizes. Make sure you are setting the `Object_hw` and `Object_hh` fields in `rock_new` as explained previously.

Now go to `abc123_proj2.asm` and in the `game_normal` function, uncomment the 4 lines of code after the "check for win condition" line. Now, when you destroy all the rocks, the game ends! Woo! (And if you wait 5 seconds, the game restarts.)

Okay, but you're not supposed to destroy the rocks by driving into them, are you...?

Shooting the rocks

Each rock calls `rock_collide_with_bullets` in its `rock_update` function. (It does, doesn't it? Check that! And make sure you're passing the same rock object that was passed to `rock_update`.)

`rock_collide_with_bullets(rock)` will work like this:

- loop over all the objects in the `objects` array.
  - Have a look at `Object_delete_all` in `object.asm`. You can use the same `for` loop pattern.
  - Of course, the stuff *inside* the loop will be different.
- when it finds an object whose `Object_type` field is `TYPE_BULLET`...
  - it will use `Object_contains_point(obj, x, y)` to see if **the bullet's x/y coordinates are inside the rock.**
  - if so:
    - call `rock_get_hit` on the rock
    - call `Object_delete` on the bullet
    - exit the loop and return from the function.
  - if not, keep looping! don't exit after the first bullet. you want to check ALL the bullets.

Now you should be able to shoot bullets to destroy them!

Checkpoint: You now have about a 70-75%. Not too much more now...

Rocks of different sizes

Right now, the rocks just disappear. That's not right.

1. Extend `rock_new` to use its third argument, the type.
    - **Use the right bounding box constants depending on the type argument:**
        - `TYPE_ROCK_L` should use `ROCK_L_HW` and `ROCK_L_HH`
        - `TYPE_ROCK_M` should use `ROCK_M_HW` and `ROCK_M_HH`
        - `TYPE_ROCK_S` should use `ROCK_S_HW` and `ROCK_S_HH`
    - Also, change the **velocity** depending on the type:
        - `TYPE_ROCK_L` should use `ROCK_VEL`
        - `TYPE_ROCK_M` should use `ROCK_VEL * 4`
        - `TYPE_ROCK_S` should use `ROCK_VEL * 12`
2. Call `jal rock_get_hit` from `rock_collide_m` and `rock_collide_s` as well.
3. Fill in `rock_draw_m` and `rock_draw_s` the same way you did `rock_draw_l`.
    - Use `spr_rock_m` and `spr_rock_s` respectively.
4. Finally, expand `rock_get_hit`.
    - If the rock is large...
        - Use `rock_new` twice to create two medium rocks **at the same position.**
    - If the rock is medium...
        - Use `rock_new` twice to create two small rocks **at the same position.**
    - And at the end, just like before, **delete the old rock.**

Now **big rocks should split into two, and medium rocks should split into**

two. This should work both when you shoot them and when you run into them!

> Sometimes, if you run into them, it might seem like they don't split. That's because the new rocks can immediately run into the player and be destroyed. But don't worry! We'll fix that.

Damaging the player

You're actually almost done with the rocks!! Let's make them hazardous so your ship is no longer an unstoppable rock vacuum. There is a `player_damage(int amount)` function which you can use to hurt the player.

In the `rock_collide_*` methods, **after the call to `rock_get_hit`,** call `player_damage` with the following arguments:

- `rock_collide_l` should call `player_damage(3)`
- `rock_collide_m` should call `player_damage(2)`
- `rock_collide_s` should call `player_damage(1)`

Now let's go back to `player.asm` and implement `player_damage`. **First, let's make sure it's being called correctly.** Put this inside `player_damage`:

```
syscall_print_int
println_str " damage"
```

This will print the amount of damage when the function is called. Now run into the rocks. It should print 3 damage when you hit large ones, 2 when you hit medium ones, and 1 when you hit small ones. **If it doesn't, fix your code!!**

Once you're sure it's working, you can remove those prints.

Here's the idea of damaging the player:

- The player starts with 5 health. **That's what the blue and white thing at the bottom of the screen is showing.**
- When they get damaged...
  - The damage amount is subtracted from their health.
  - When their health reaches 0, **they disappear and lose a life.**
  - Then, after a short delay:
    - If they still have lives left, they will **respawn** (revive, resurrect, reappear, whatever).
    - If they don't have lives left, it's **game over!**

So let's start with the easy stuff in `player_damage`.

- Subtract the argument from `player_health`. **Don't forget to store!**
  - *Tip: you can use the `maxi` macro to prevent a value from going below 0. For example:*

```
sub  t0, t0, a0
maxi t0, t0, 0  # t0 = max(t0, 0)
```

- If `player_health` reaches 0...
  - Decrement `player_lives`. (Use `maxi` to prevent it from going below 0 again.)

Test it out. What happens? Well, your ship is still a rock vacuum, but **your displayed health and lives should be changing.** Success?

Invulnerability frames

Long ago, video game designers realized that hurting the player 60 times a second was a total dick move. So, they invented **invulnerability frames** or "iframes" for short. When the player is hurt, they become temporarily invulnerable for a short period of time. There's a `player_iframes` frame timer variable for you.

Change the `player_damage` logic to look like this:

```
if(player_iframes == 0) { // new!
    damage the player like before

    if(player_health == 0)
        lose a life like before
    else
        player_iframes = PLAYER_HURT_IFRAMES; // new!
}
```

Now test it out. When you run into a rock, you start flashing! And you can't get hit anymore!..... Wait, you're still flashing! **You never stop flashing! You just invented the invulnerability powerup??**

The problem is it's a **frame timer,** but you **never decremented it.**

Have a look at `player_collide_all`. Notice it checks for `player_iframes` and just returns if the player is invulnerable. That's how this works!

In `player_update`, add some code to decrement `player_iframes` if it's nonzero, just like you did with `player_fire_time`. Now when you get hurt, you will flash for a second, and then go back to normal.

Almost there.

Disappearing and respawning

In `player_damage`, in the same part where you decrement `player_lives`, put some code to do:

```
player_deadframes = PLAYER_RESPAWN_TIME;
```

Now when you run out of health, you disappear! ..... kind of. Actually, **you're a ghost now.** You can still fire bullets and move around. That's weird. You

have to do **the other part.**

So now change `player_update` like so:

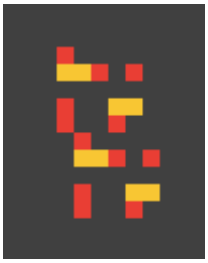
- If `player_deadframes == 0`, **update as normal.** Otherwise...
  - Decrement `player_deadframes`.
  - If `player_deadframes` is not 0, **just return from `player_update`.**
    - This will prevent the user from being able to move, shoot etc.
  - Otherwise, that means the frame timer is up, and it's time to respawn the player!
    - If `player_lives > 0`:
      - Call `player_respawn` and set `player_iframes` to `PLAYER_RESPAWN_IFRAMES`.
    - Else, call `lose_game` and **return from `player_update`.**

And that's it!! **The game is done!!** You can now lose all your lives and get a game over, or destroy all the rocks and get a congratulation message. :D

Checkpoint: You now have a 90%!! You're so close!

But wait, there's one more thing to do: EXPLODE. Wait, no! Not you!! The rocks and spaceship!

The Explosions



The **explosions** are a fun animated effect shown when rocks or the player are destroyed. They don't change how the game works, but they make it look nicer :)

I've made a short 6-frame explosion animation for you. All the

explosion objects do is show this animation.

In `explosion.asm`...

- `explosion_new(x, y)` should make a new `TYPE_EXPLOSION` object, just like `bullet_new` and `rock_new`.
  - Set its x/y to the arguments.
  - Set its hw/hh to `EXPLOSION_HW/HH`.
  - It has two extra fields, similar to bullets.
    - Set its `Explosion_timer` field to `EXPLOSION_ANIM_DELAY`.
    - Set its `Explosion_frame` field to 0.
- `explosion_draw` should use `Object_blit_5x5_trans`.
  - The second argument is `spr_explosion_frames[explosion.Explosion_frame]`.
    - **Note: if your explosions look like rainbow garbage,** you probably forgot to load the value after calculating the array address.
  - `spr_explosion_frames` is in `graphics.asm`. It's just a `.word` array.

Now call `explosion_new` in two places:

- In `player_damage`, when the player's health reaches 0, using the player's x/y as arguments;
- In `rock_get_hit`, **BEFORE calling `Object_delete` !!**

Now when you blow up rocks or your ship, ... a yellow dot appears. That's cause the explosion object is not animating. So, **in `explosion_update`:**

- Decrement the `Explosion_timer` field. If it's 0:
  - Set the `Explosion_timer` field back to `EXPLOSION_ANIM_DELAY`.
  - Increment the `Explosion_frame` field. If it's >= 6:
    - Delete the object.

And that... is it. You have a 100%.