Project 2 Engine

Reference for the code I've given you

Objects

Each "thing" in the game is represented as an **Object** `struct` with these fields:

- `Object_type` - one of the `TYPE` constants
- `Object_x` - (24.8) x position
- `Object_y` - (24.8) y position
- `Object_vx` - (24.8) x velocity
- `Object_vy` - (24.8) y velocity
- `Object_hw` - (24.8) half-width (measured from center)
- `Object_hh` - (24.8) half-height
- And then, up to 5 more fields after these (which some objects use for extra variables).

Notice all the numerical fields are marked **(24.8)** - these are **fixed-point** numbers with 8 fractional bits.

Position, width, and height ¶

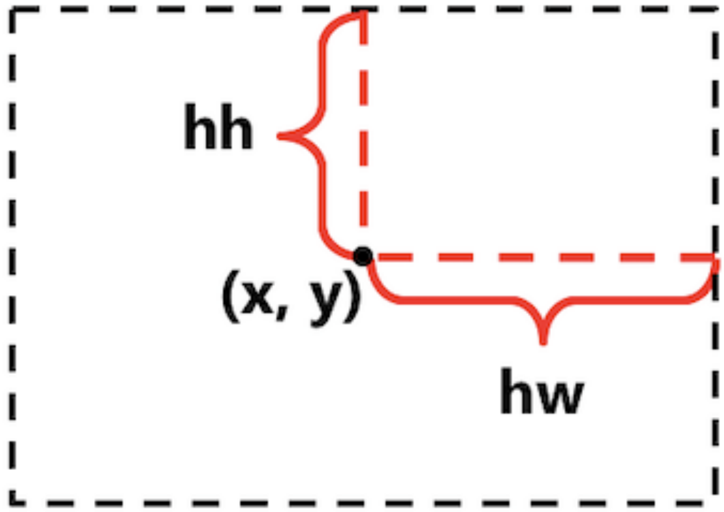
The x and y coordinates of an object are at its **center.**

The **half-width** and **half-height** define its width and height as measured **from the center.** Think of it like the rectangular analog of a circle's radius.

The top of the object is at `y - hh`; the bottom at `y + hh`; the left side at `x - hw`; and the right side at `x + hw`.

These four sides define the object's **bounding box,** which is used to perform **collision** (detecting when one object is touching another).

Also, the `Object_blit_5x5_trans` function draws the image at the **top-left** of the bounding box.



Velocity

The **velocity** `(vx, vy)` is what will be added to the position by the `Object_accumulate_velocity` function. And that's how objects move!

The `Object_apply_acceleration` function adds an acceleration vector to these fields.

Type

The `Object_type` field says what kind of object it is - a player, a rock, or whatever. The type constants I've given you are:

- `TYPE_EMPTY` (aka 0): an **inactive object.** It will not be updated, drawn, or collided with.
- `TYPE_PLAYER` - the player
- `TYPE_BULLET` - a bullet the player shoots
- `TYPE_ROCK_L` - large rock
- `TYPE_ROCK_M` - medium rock
- `TYPE_ROCK_S` - small rock
- `TYPE_EXPLOSION` - explosion animation

The Object array

There is a global array (see `globals.asm`) of objects called... `objects`. What did you expect? ;)

There are `MAX_OBJECTS` objects in the array. This defaults to 50.

`objects[0]` can also be accessed as `player`. For example:

```
    la t0, player
    lw t1, Object_x(t0) # t1 = player.x
```

Allocating and deallocating Objects

To allocate (create) a new Object, call `Object_new` with the object type as its argument. **This function can return 0 (null) if there's no more space in the `objects` array.** It's important to check its return value, and only use it if it's not 0. For example:

```
    li  a0, TYPE_BULLET
    jal Object_new
    # check if it's null!
    beq v0, 0, _no_new_bullet
        # here, it's not null, so we can access v0.
    _no_new_bullet:
```

When an object is no longer needed, call `Object_delete` with the object pointer as its argument:

```
    move a0, s0 # say s0 has "the current object" in it
    jal Object_delete
```

Object methods

This engine uses a limited form of Object-Oriented Programming. It does this by using the objects' `type` field to decide which function to call in some cases.

For all the methods, the `this` object is passed as `a0`.

`Object_update_all()` will call each (non-empty) object's `update` method. These methods are listed in `globals.asm`, in the `object_update_funcs` array.

`Object_draw_all()` will call each (non-empty) object's `draw` method, if it has one. These methods are listed in `globals.asm`, in the `object_draw_funcs` array.

`player_collide_all()` will test for collision between the player and any object that has a non-null entry in the `player_collide_funcs` array. If a collision with the player occurs, that method is called.

`objects.asm` reference

Here are the functions provided in `objects.asm`.

Signature	Description
`Object_update_all()`	calls the `update` method on all non-empty objects.
`Object_draw_all()`	calls the `draw` method on all non-empty objects.
`Object_delete_all()`	delete all objects in the `objects` array (clean slate!)
`Object* Object_new(type)`	allocates a new Object from the `objects` array, and returns its address; or returns null if there are no more empty slots in the `objects` array.
`Object_delete(obj)`	mark the given object as empty, and zeroes out its other fields as well.
`Object_accumulate_velocity(obj)`	add the given object's `vx` and `vy` fields to its `x` and `y` fields.
`Object_apply_acceleration(`	adds `ax` to the object's `vx`, and `ay` to

`ax, ay)`	the object's `vy`.
`Object_wrap_position(obj)`	wraps the object's position to the range `[0.0, 64.0)`.
`Object_damp_velocity(obj, damping)`	*divides* the object's velocity fields by `damping`.
`Object_blit_5x5_trans(obj, pat)`	draws the image pointed to by `pat` to the top-left of the object's bounding box. if drawn near the edges of the screen, it will wrap around the edges.
`bool Object_contains_point(obj, x, y)`	returns a boolean saying whether the point `(x, y)` is within the object's bounding box.
`bool Objects_overlap(obj1, obj2)`	returns a boolean saying whether the two objects' bounding boxes overlap.

`macros.asm` reference

Here are the macros provided in `macros.asm`.

Syntax	Description
`lstr t0, "hello!"`	Puts a string into the `.data` segment and loads its address (using `la`) into the register.
`print_str "hello!"`	Prints the string to the console.
`println_str "hello!"`	Prints the string to the console, and then prints a newline.
`newline`	Prints a newline to the console.
`inc t0`	Adds 1 to a register.
`dec t0`	Subtracts 1 from a register.
`min t0, t1, t2`	Sets `t0` to the smaller of registers `t1` and `t2`.

`mini t0, t1, 10`	Sets `t0` to the smaller of `t1` and the constant 10.
`max t0, t1, t2`	Sets `t0` to the larger of registers `t1` and `t2`.
`maxi t0, t1, 10`	Sets `t0` to the larger of `t1` and the constant 10.
`enter [s0, ...]`	Pushes `ra` and any `s` registers you list after it. Comes at the beginnings of functions. **I introduced these in lab 6.**
`leave [s0, ...]`	Pops any `s` registers and `ra`, then returns with `jr ra`. Comes at the ends of functions.
`syscall_print_int`	Shorthand for `li v0, 1` and `syscall`. Trashes `v0`. All the `syscall_` macros below do the same.
`syscall_print_float`	*syscall 2*
`syscall_print_double`	*syscall 3*
`syscall_print_string`	*syscall 4*
`syscall_read_int`	*syscall 5*
`syscall_read_float`	*syscall 6*
`syscall_read_double`	*syscall 7*
`syscall_read_string`	*syscall 8*
`syscall_exit`	*syscall 10*
`syscall_print_char`	*syscall 11*
`syscall_read_char`	*syscall 12*
`syscall_time`	*syscall 30*
`syscall_midi_out`	*syscall 31*
`syscall_sleep`	*syscall 32*
`syscall_midi_out_sync`	*syscall 33*
`syscall_print_hex`	*syscall 34*
`syscall_print_bin`	*syscall 35*
`syscall_print_uint`	*syscall 36*

`syscall_seed_rand`	*syscall 40*
`syscall_rand_int`	*syscall 41*
`syscall_rand_range`	*syscall 42*

`math.asm` reference

Here are the functions provided in `math.asm`. You probably won't need to use many of these, but they're there if you need them!

Signature	Description
`int random(int x)`	returns a random integer in the range `[0, x - 1]`. So, `random(100)` will return a maximum of 99.
`int clamp(int val, int lo, int hi)`	if `val < lo`, returns `lo`; else if `val > hi`, returns `hi`; else returns `val`.
`f16 sin(int angle)`	takes an integer angle in *degrees* and returns its sine as a `16.16` fixed-point number.
`f16 cos(int angle)`	takes an integer angle in *degrees* and returns its cosine as a `16.16` fixed-point number.
`(f16, f16) sin_cos(int angle)`	takes an integer angle in *degrees* and returns both the sine (in `v0`) and cosine (in `v1`) as `16.16` fixed-point numbers.
`(f24, f24) to_cartesian(f24 r, int t)`	Converts a polar coordinate `(r, t)` to cartesian coordinate `(x, y)` returned as `(v0, v1)`. The radius and return values are `24.8` fixed-point numbers.
`f24 hypot(f24 dx, f24 dy)`	returns `√(dx^2 + dy^2)` (the length of the hypotenuse). all values are `24.8` fixed-point numbers.
`(f24, f24) normalize_24_8(f24`	given a vector `(x, y)`, normalizes its

`x, f24 y)`	magnitude to 1. all values are `24.8` fixed-point numbers.
`f16 sqrt_16_16(f16 x)`	given a `16.16` fixed-point number, returns its square root.
`f16 rsq_16_16(f16 x)`	given a `16.16` fixed-point number, returns its *reciprocal* square root (i.e. `1/√x`).

© 2016-2020 Jarrett Billingsley