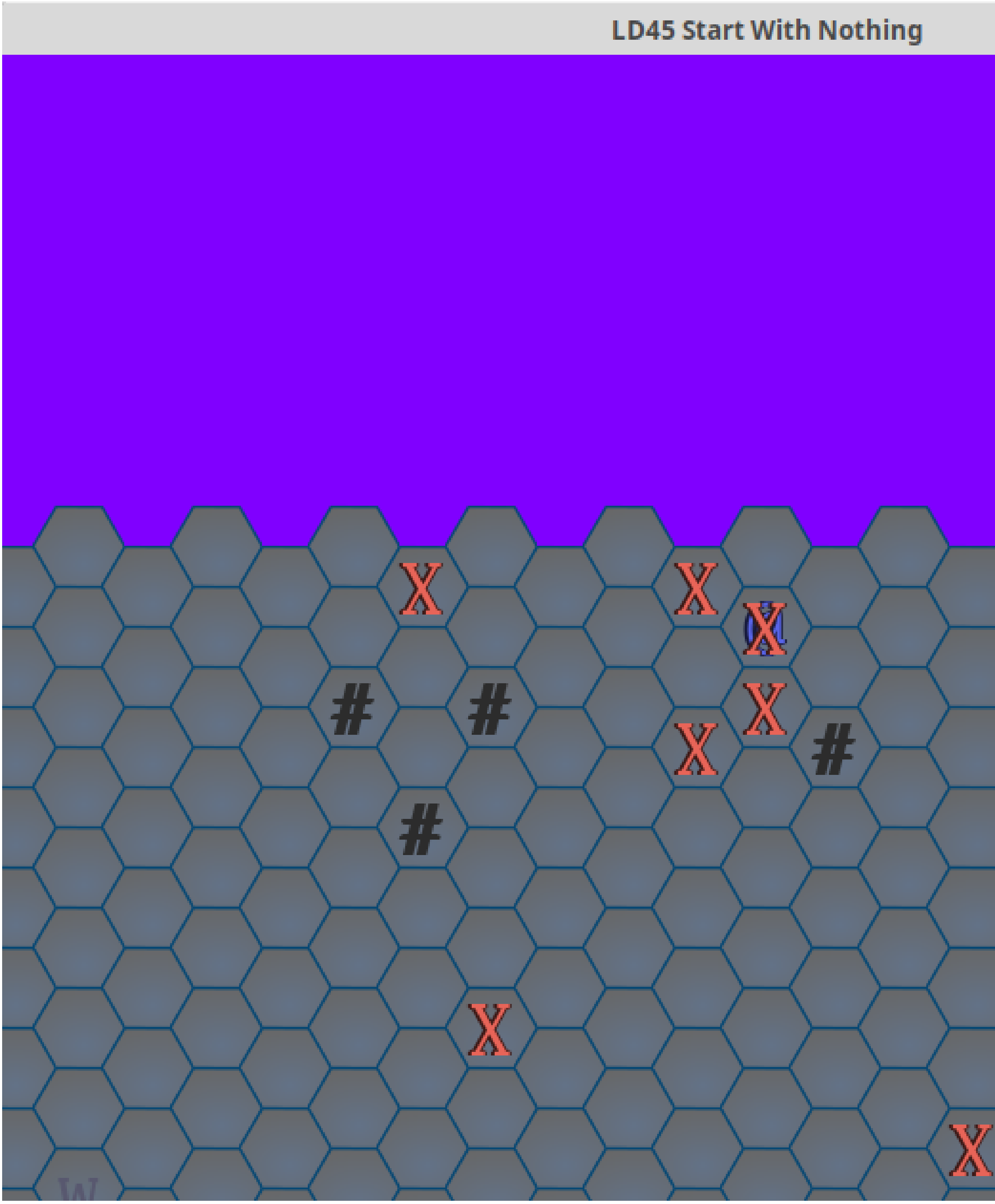

LD45 Start With Nothing

Contents

1 Introduction

This is the source code of my entry to Ludum Dare 45, which took place on the weekend of 4-6. October, 2019. Ludum Dare is a game jam, where competitors attempt to create a game according to a theme over the course of a weekend.

Long ago, the rules stated that the game needed to be written completely from scratch during the competition time. Due to the rise of freely-available game engines and the inherent difficulties in defining what "from scratch" actually means, they have considerably relaxed this requirement, and now allow most any preexisting source code to be used, while art assets must still be created inside the time limit.

A few days before the competition officially began, I started writing the framework of getting Rust and SDL talking to each other properly. At the point when the theme was announced, it was successfully opening a window, responding to events, and drawing solid-colored rectangles to the screen.

This document is written with noweb, a literate programming tool. In addition to producing this documentation, it extracts the source code from here and reassembles it into the form that the compiler expects. In this way, the code in the document and the code that actually runs the program are kept in sync. There are many hyperlinked annotations in and around the code blocks to aid navigation around the program text, and hopefully this report is organized such that reading through it from beginning to end will present everything in a logical order.

1.1 Theme & Direction

The theme this time around is "Start with nothing". Somewhat appropriate, since I'm almost starting with nothing code-wise. I think I'm going to work towards a Rogue-like game where you start with no character class, no skills, no items, and no information.[1]

[1] Depending on how development goes, you may end that way, too.

As far as setting, I'm thinking alien abduction works reasonably well. In the opening cutscene (which is probably text), you are awoken by a blinding light and then find youself in an unfamiliar environment in your pyjamas.

For the first game mechanics, I'm thinking about taking a page from the old Dalek games, where your primary means of disabling enemies is making them run into each other as they try to pursue you. If I get nothing but that working, then I know that it's still a reasonably satisfying game, and there's plenty of room to add things like map generation, environmental obstacles, puzzles, and the like after I get that core bit working.

On the technology side, I think I need to get some text rendering working first and then try to get some kind of tile-based world that you can walk around in.

1.2 Program Organization

The implementation is divided into three files: *lib, main, & test.* These are each compiled into separate crates.

Lib.rs contains the bulk of the implementation, and is divided into several modules. It gets compiled into an rlib file that can be linked with other source files to make a complete program.

4a ⟨*lib.rs* 4a⟩≡

```
pub mod sdl  { ⟨SDL: Definitions 5a⟩ }  // SDL Bindings
pub mod fut  { ⟨FUT: Definitions 21a⟩ }  // Async, Futures
pub mod sim  { ⟨SIM: Definitions 30a⟩ }  // Game Logic
pub mod main { ⟨MAIN: Definitions 4d⟩ }
```

Test.rs contains any automated testing code that I elect to write (currently none). These test must pass before the main program is built.

4b ⟨*test.rs* 4b⟩≡

```
```

Main.rs is the primary entry point for the program. In this case, it is simply dispatching to a run function in *lib*. elect to write (currently none). These test must pass before the main program is built.

4c ⟨*main.rs* 4c⟩≡

```
include!("import.inc"); use std::*;
fn main()->Result<(), Box<dyn error::Error>> { ld45::main::run() }
```

THE MAIN MODULE contains the high-level logic for running the game. Its only export is the `run` function which contains both the initialization and main game loop code.

4d ⟨*MAIN: Definitions* 4d⟩≡ (4a) 33▷

```
use std::error::*;
use crate::sdl::*;
use crate::fut::*;
use crate::sim;
use crate::sim::Command::*;

pub fn run()->Result<(), Box<dyn Error>>
{   ⟨MAIN: Initialization 7b⟩
    'mainloop: loop
    {   ⟨MAIN: Process Events 10e⟩
        ⟨MAIN: Run Simulation 30d⟩
        ⟨MAIN: Draw Frame 32b⟩
    }
    Ok(())
}
```

2 LibSDL2

THE SDL MODULE is responsible for communicating with `libsdl2`, which is ultimately responsible for the graphics, sound, and input processing. There is a pre-existing rust crate for this, but I have elected to write my own.

5a ⟨*SDL: Definitions 5a*⟩≡ (4a) 6a ▷

```
use std::ffi::*;
use std::os::raw::*;
use std::sync::*;
use std::*;
use std::error::Error;
use std::marker::*;
use std::ptr::*;

⟨SDL: C Types 5c⟩
#[link(name="SDL2")] extern { ⟨SDL: C Functions 5b⟩ }
```

2.1 SDL Error Handling

The first order of business is to be able to detect and respond to SDL errors before they become segmentation faults. SDL provides one function for this:

5b ⟨*SDL: C Functions 5b*⟩≡ (5a) 6c ▷

```
fn SDL_GetError() -> *const c_char;
```

Like many C libraries, SDL often uses a sentinel value (such as a null pointer) to indicate a fault has occurred, and other values are useful results. The `IsErr` trait is designed to handle this situation; it has a single method that will distinguish erroneous values from legitimate ones.

`ErrCheck` is a transparent wrapper around any type that implements `IsErr`. Its `unwrap` method transforms SDL's error strategy into Rust's `Result` types, which can then be handled the same way as any other Rust error. `ErrCheck` is private to encourage implementations to turn it into an `SdlError` as soon as possible, given that the description of an SDL error is lost as soon as another error occurs.

5c ⟨*SDL: C Types 5c*⟩≡ (5a) 6b ▷

```
// Error Handling Types
trait IsErr { fn is_err(&self)->bool; }
#[derive(Debug)] #[repr(transparent)] struct ErrCheck<T:IsErr>(T);
impl<T:IsErr> ErrCheck<T>
{   fn unwrap(self)->Result<T,SdlError>
    {   if self.0.is_err() { Err(SdlError::fetch()) }
        else               { Ok(self.0)            } } }
```


The `SdlError` class holds its own copy of SDL's error description string, so it will retain its identity even if anothe SDL error occurs. The only way to create one is via `fetch()`, which will take the information about the most recent error from SDL.

6a ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁5a 7a▷

```
#[derive(Debug, Clone)] pub struct SdlError( String );
impl SdlError
{   fn fetch()->SdlError
    {   SdlError( unsafe { CStr::from_ptr(SDL_GetError())
                           .to_string_lossy()
                           .into_owned()              } ) } }

impl Error for SdlError {}

impl fmt::Display for SdlError
{   fn fmt(&self, f: &mut fmt::Formatter<'_>)
    -> fmt::Result { write!(f, "{:?}", self) } }
```

Often, an SDL function will return a negative integer to indicate that an error has occurred, and other values represent a successful result. This logic is encapsulated by implementing `IsErr for c_int`.

6b ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁5c 8b▷

```
impl IsErr for c_int { fn is_err(&self)->bool { *self > 0 } }
```

2.2 *SDL Initialization*

Before doing anything, SDL needs to be initialized using the `SetMainReady` and `Init` functions. It also may hold resources that won't be cleaned up efficiently by the operating system when the process exits, and so expects `Quit` to be explicitly called before program exit.

6c ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁5b 8a▷

```
// Initialization and cleanup
fn SDL_SetMainReady();
fn SDL_Init(flags: u32)->ErrCheck<c_int>;
fn SDL_Quit();
```

As Rust uses the Resource Acquisition Is Initialization (RAII) pattern extensively, the most natural way to accomplish these requirements is to have a context object that calls `Quit` when it is dropped.

7a ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁ 6a 10a ▷

```
#[derive(Debug)] pub struct SDL { guard: MutexGuard<'static, ()> }
impl SDL { ⟨SDL: SDL Methods 7c⟩ }
impl Drop for SDL
{ fn drop(&mut self) { unsafe { SDL_Quit(); } } }
```

This allows the program to perform all of the necessary startup and cleanup tasks with a single line of code:

7b ⟨*MAIN: Initialization* 7b⟩≡ (4d) 9a ▷

```
let sdl = SDL::init()?;
```

SDL generally expects all calls to be made from the same thread and for there to only ever be one SDL instance initialized at a time. We therefore set up a static mutex to ensure that only one `SDL` object is ever alive at one time.

7c ⟨*SDL:* SDL *Methods* 7c⟩≡ (7a) 8c ▷

```
pub fn init()->Result<SDL, Box<dyn Error>>
{   let guard = ⟨SDL: Enforce Singleton 7d⟩;
    let flags = 0x20; //SDL_VIDEO
    unsafe { SDL_SetMainReady();
             SDL_Init(flags).unwrap()?; }
    Ok(SDL { guard })
}
```

Because static initializers for mutexs are still unstable, the code necessary to set up the mutex is slightly involved. The static variable holding the mutex must be mutable, which forces an `unsafe` block, and we need to use `sync::Once` to ensure that only one mutex is ever created.

7d ⟨*SDL: Enforce Singleton* 7d⟩≡ (7c)

```
unsafe
{   static mut MUTEX        : Option<Mutex<()>> = None;
    static     MUTEX_EXISTS : Once              = Once::new();

    MUTEX_EXISTS.call_once(|| { MUTEX = Some(Mutex::new(())); });
    MUTEX.as_ref().unwrap()
}.try_lock()?
```

2.3 Window Creation

Creating a window that we can draw on is fairly straightforward, except for the matter of determining appropriate flags. I'll defer that investigation until later.

8a ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁6c 9b▷

```
fn SDL_CreateWindow<'sdl>
( title: *const c_char,
  x: c_int, y: c_int,
  w: c_int, h: c_int,
  flags: u32
) -> ErrCheck<Window<'sdl>>;

fn SDL_DestroyWindow(win: WindowHandle);
```

Like the SDL context itself, it makes sense to use RAII to automatically destroy any window that Rust no longer has a handle to. `CreateWindow` might also fail ane return a null pointer, so we define `IsErr` to handle that case.

8b ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁6b 9c▷

```
#[derive(Debug, Copy, Clone)]
#[repr(transparent)]     struct WindowHandle(*const c_void);

#[derive(Debug)]
#[repr(transparent)] pub struct Window<'sdl>(WindowHandle,
                                                PhantomData<&'sdl SDL>);

impl<'sdl> IsErr for Window<'sdl> { fn is_err(&self)->bool {
    (self.0).0.is_null() }}

impl<'sdl> Drop for Window<'sdl> { fn drop(&mut self) {
    unsafe { SDL_DestroyWindow(self.0); }}}

impl<'sdl> Window<'sdl> { ⟨SDL: Window Methods 13b⟩ }
```

We expose a safe wrapper for the native function:

8c ⟨*SDL:* SDL *Methods* 7c⟩+≡ (7a) ◁7c 10d▷

```
pub fn create_win<'sdl>(&'sdl self, title:&str, pos:(i32,i32), size:(i32,i32))
-> Result<Window<'sdl>, Box<dyn Error>> { unsafe {
    let c_title = CString::new(title)?;
    Ok(SDL_CreateWindow(c_title.as_ptr(),
                        pos.0,  pos.1,
                        size.0, size.1,
                        0                ).unwrap()?) }}
```

And the actual initialization is a one-line call again:

9a ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁ 7b 13c ▷

```
let win = sdl.create_win("LD45- Start With Nothing",
                         (0, 0), (800, 600))?;
```

2.4 Event Polling

Input is handled through SDL's event system. There are two functions here that we're interested in right now. `PumpEvents` tells SDL to check all of the various devices for new input, and enqueues events for anything that's happened. `PollEvent` takes the first event off of the event queue and returns it, or an indication that the queue is empty.

9b ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁ 8a 12d ▷

```
fn SDL_PumpEvents();
fn SDL_PollEvent(ev:*mut SdlEventUnion)->c_int;
fn SDL_Delay(ms:u32);   fn SDL_GetTicks()->u32;
```

Events in SDL are represented by a tagged union. The first field is an integer specifying the type of the event, and the rest of the space has different layouts depending on what that field says.

9c ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁ 8b 9d ▷

```
#[repr(C)]
union SdlEventUnion { event_type: u32,
                      bytes: [u8;56],
                      ⟨SDL: SdlEventUnion fields 9e⟩ }
```

In addition to the event type, all of the structures have a common preamble, so we can use that in the case we have an unknown event occur.

9d ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁ 9c 12a ▷

```
#[repr(C)] #[derive(Debug,Copy,Clone)]
pub struct SdlCommonEvent { pub event_type: u32,
                            pub timestamp:  u32 }
```

9e ⟨*SDL:* `SdlEventUnion` *fields* 9e⟩≡ (9c) 11d ▷

```
common: SdlCommonEvent,
```

Rust's equivalent to a tagged union is an enum with data-containing variants. We define `SdlEvent` to represent all of the possible event types, with one variant for each type id.

10a ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁ 7a 10b ▷

```
#[derive(Debug, Copy, Clone)]
pub enum SdlEvent { Other(SdlCommonEvent),
                    ⟨SDL: SdlEvent Variants 11a⟩ }
```

To facilitate conversions from the C union into the Rust enum, we define the `From` trait:

10b ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁ 10a 10c ▷

```
impl From<SdlEventUnion> for SdlEvent
{   fn from(raw: SdlEventUnion) -> SdlEvent { unsafe
    {   match raw.event_type { ⟨SDL: SdlEvent Dispatch Table 11b⟩
                               _ => SdlEvent::Other(raw.common) }
    }}
}
```

Usually, you'll want to process all pending events in an event loop embedded inside the main game loop. The `iter_events` method wraps `SDL_PollEvents` to enable its use in a Rust `for` loop control statement.

10c ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁ 10b 14a ▷

```
pub struct SdlPendingEventsIter<'a> { _sdl: &'a SDL }
impl<'a> Iterator for SdlPendingEventsIter<'a>
{   type Item = SdlEvent;
    fn next(&mut self) -> Option<SdlEvent>
    {   let mut result = SdlEventUnion { event_type: 0 };
        let code = unsafe
        { SDL_PollEvent(&mut result as *mut SdlEventUnion) };
        if code > 0 { Some(SdlEvent::from(result)) }
        else        { None                         } } }
```

10d ⟨*SDL:* SDL *Methods* 7c⟩+≡ (7a) ◁ 8c 16a ▷

```
pub fn iter_events<'a>(&'a self) -> SdlPendingEventsIter<'a>
{   unsafe { SDL_PumpEvents() };
    SdlPendingEventsIter { _sdl: self } }
pub fn delay(&self, ms:u32) { unsafe { SDL_Delay(ms); }; }
pub fn ticks(&self) -> u32  { unsafe { SDL_GetTicks() } }
```

10e ⟨*MAIN: Process Events* 10e⟩≡ (4d)

```
sdl.delay(10);
for e in sdl.iter_events() {
    match e { ⟨MAIN: Event Handlers 11c⟩
              event@_ => { println!("Unhandled event: {:?}",
                                    event);                  } } }
```

2.5 *Quit Event*

The first and most basic event that we want to handle is `SDL_QUIT`. It is fired when the user attempts to close the window or otherwise use operating system facilities to ask the program to stop. It has no fields other than what's defined in the common preamble, so we just need to make an `SdlEvent` variant for it and add its id to the dispatch table.

11a ⟨*SDL:* `SdlEvent` *Variants* 11a⟩≡ (10a) 12b ▷

```
Quit(SdlCommonEvent),
```

11b ⟨*SDL:* `SdlEvent` *Dispatch Table* 11b⟩≡ (10b) 12c ▷

```
0x100 => SdlEvent::Quit(raw.common),
```

As far as handling it in the main program, the simplest choice is to simply break the game loop and let `main` return. In a more complicated game/program, you would likely want to ask for confirmation or attempt to save progress before exiting.

11c ⟨*MAIN: Event Handlers* 11c⟩≡ (10e) 36a ▷

```
SdlEvent::Quit(_) => { break 'mainloop; }
```

2.6 *Keyboard Events*

The keyboard is the most controller-like device attached to most computers. There's a lot of complicated parts to this that I'm going to completely ignore, as I only want to be able to detect a few button presses and don't care about text input. There is a couple of structs that we need to bring in from the C header files for this, and I need to rely on the convention that C enums are always represented as integers.

11d ⟨*SDL:* `SdlEventUnion` *fields* 9e⟩+≡ (9c) ◁ 9e

```
kbd: SdlKeyboardEvent,
```

12a ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁ 9d 13a ▷

```
#[repr(C)] #[derive(Debug,Copy,Clone)]
pub struct SdlKeyboardEvent { pub event_type: u32       ,
                              pub timestamp : u32       ,
                              pub windowID  : u32       ,
                              pub state     : u8        ,
                              pub repeat    : u8        ,
                                  padding2  : u8        ,
                                  padding3  : u8        ,
                              pub sym       : SdlKeysym }
#[repr(C)] #[derive(Debug,Copy,Clone)]
pub struct SdlKeysym { pub scancode : c_int,
                       pub keycode  : c_int,
                       pub modifiers: u16  ,
                           unused   : u32  }
```

Because the interesting information (which key was pressed) is buried quite deep in this structure, I'll duplicate the scancode in the `SdlEvent` variants at the top level, so that pattern matching works a little easier. We want the scancode rather than the keycode because it indicates the keyboard position rather than the letter that key represents.

12b ⟨*SDL:* `SdlEvent` *Variants* 11a⟩+≡ (10a) ◁ 11a

```
KeyUp(i32, SdlKeyboardEvent), KeyDn(i32, SdlKeyboardEvent)
```

12c ⟨*SDL:* `SdlEvent` *Dispatch Table* 11b⟩+≡ (10b) ◁ 11b

```
0x301 => SdlEvent::KeyUp(raw.kbd.sym.scancode, raw.kbd),
0x300 => SdlEvent::KeyDn(raw.kbd.sym.scancode, raw.kbd),
```

2.7 Getting a Renderer

We'll be using the accelerated 2D renering facilities from SDL, which are accessed via an `SDL_Renderer` object. It needs to be created and destroyed in much the same way as the window object was:

12d ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁ 9b 13d ▷

```
fn SDL_CreateRenderer<'win>
(     win: WindowHandle,
    index: c_int,
    flags: u32            ) -> ErrCheck<Renderer<'win>>;
fn SDL_DestroyRenderer
( handle: RendererHandle ) -> ();
```

13a ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁ 12a 14c ▷

```
#[derive(Debug, Copy, Clone)] #[repr(transparent)]
struct RendererHandle(*const c_void);

#[derive(Debug)] #[repr(transparent)]
pub struct Renderer<'win>(RendererHandle,
                          PhantomData<&'win Window<'win>>);

impl<'win> IsErr for Renderer<'win> { fn is_err(&self)->bool {
    (self.0).0.is_null() }}

impl<'win> Drop for Renderer<'win> { fn drop(&mut self) {
    unsafe { SDL_DestroyRenderer(self.0); }}}

impl<'win> Renderer<'win> { ⟨SDL: Renderer methods 14b⟩ }
```

The default paramaters seem fine, so creating the renderer is a simple call on the `Window` instance:

13b ⟨*SDL:* `Window` *Methods* 13b⟩≡ (8b)

```
pub fn create_renderer(&self) -> Result<Renderer, Box<dyn Error>>
{ Ok(unsafe { SDL_CreateRenderer(self.0, -1, 0).unwrap()? }) }
```

13c ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁ 9a 17c ▷

```
let renderer = win.create_renderer()?;
```

2.8 Drawing a Frame

SDL Makes no guarantees about the contents of the drawing buffer after a frame is presented, so best practice is to clear it before drawing. We also need to call `SDL_RenderPresent` once we're done drawing the frame.

13d ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁ 12d 14d ▷

```
fn SDL_SetRenderDrawColor
(   handle: RendererHandle,
    r:u8, g:u8, b:u8, a:u8  ) -> ErrCheck<c_int>;
fn SDL_RenderClear
(   handle: RendererHandle  ) -> ErrCheck<c_int>;
fn SDL_RenderPresent( handle: RendererHandle  );
```

Since RAII is the Rust way, we'll define a `Canvas` object that represents a single frame as it's being drawn. When that object goes out of scope, we'll use the `Drop` trait to send the completed frame to the screen.

14a ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁10c 16d▷

```
pub struct Canvas<'r, 'w:'r>(&'r Renderer<'w>);
impl<'r,'w:'r> Drop for Canvas<'r,'w> { fn drop(&mut self) {
    unsafe { SDL_RenderPresent((self.0).0) };
}}
impl<'r,'w:'r> Canvas<'r,'w> { ⟨SDL: Canvas methods 15a⟩ }
```

14b ⟨*SDL:* Renderer *methods* 14b⟩≡ (13a) 17b▷

```
pub fn start_frame<'r>(&'r self, clear_color: (u8,u8,u8))
-> Result<Canvas<'r,'win>, Box<dyn Error>>
{   unsafe { SDL_SetRenderDrawColor(self.0,
                                    clear_color.0,
                                    clear_color.1,
                                    clear_color.2,
                                    255           ).unwrap()?;
             SDL_RenderClear       (self.0        ).unwrap()?; }
    Ok(Canvas(self)) }
```

2.9 *Rectangles*

SDL uses a `Rect` struct to designate areas of the screen for various purposes, so we should probably get those working. The simplest API to test with is probably a rectange-filling function, so we'll implement that on the canvas too.

14c ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁13a 15d▷

```
#[derive(Debug, Copy, Clone)] #[repr(C)]
pub struct Rect { pub x: c_int, pub y: c_int,
                  pub w: c_int, pub h: c_int }
```

14d ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁13d 15c▷

```
fn SDL_RenderFillRects(r    : RendererHandle,
                       rects: *const Rect,
                       count: c_int          )->ErrCheck<c_int>;
```

15a ⟨*SDL:* Canvas *methods* 15a⟩≡ (14a) 18b ▷

```
pub fn set_color(&self, r:u8, g:u8, b:u8, a:u8)
-> Result<(), Box<dyn Error>>
{ Ok( unsafe { SDL_SetRenderDrawColor((self.0).0, r, g, b, a)
               .unwrap()?;
             })}
pub fn fill_rects(&self, rects: &Vec<Rect>)
-> Result<(), Box<dyn Error>>
{ Ok( unsafe { SDL_RenderFillRects((self.0).0,
                                   rects.as_ptr(),
                                   rects.len() as i32).unwrap()?; })}
```

15b ⟨ *(removed)* fill_rects *example* 15b⟩≡

```
canvas.set_color(200,200,200,255)?;
canvas.fill_rects(&vec![ Rect { x: 10, y: 10, w:140, h:80 },
                         Rect { x:170, y: 10, w:140, h:80 },
                         Rect { x:170, y:110, w:140, h:80 }, ])?;
```

2.10 *Asset Loading*

Most of SDL's asset loading functions (and those in its partner libraries) can use an RWOp structure to read data.

I had to remove the close call because it was making SDL_TTF segfault. Apparently, it needs to stay open the entire time the font object exists which the documentation never mentions. Given that the actual data is statically linked into the progam anyway, this should be only a tiny amount of memory leakage and only during initialization.

15c ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁ 14d 16c ▷

```
fn SDL_RWFromConstMem<'a>(mem : *const u8,
                          size: c_int     )->ErrCheck<RWops<'a>>;
```

15d ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁ 14c 16b ▷

```
#[repr(transparent)] #[derive(Debug,Copy,Clone)]
struct RWopsHandle(*mut RWopsHeader);

#[repr(transparent)] #[derive(Debug)]
pub struct RWops<'sdl>(RWopsHandle, PhantomData<&'sdl SDL>);
//impl<'sdl> Drop for RWops<'sdl> { fn drop(&mut self) { unsafe {
//    let handle:RWopsHandle = self.0;
//    if !handle.0.is_null()
//    { ((*(handle.0)).close)(handle); }}}}
impl<'sdl> IsErr for RWops<'sdl> { fn is_err(&self)->bool {
    (self.0).0.is_null() }}

#[repr(C)]
struct RWopsHeader {
    size: extern fn (RWopsHandle)->i64,
    seek: extern fn (RWopsHandle, i64, c_int)->i64,
    read: extern fn (RWopsHandle, *mut c_void, isize, isize)->isize,
    write: extern fn (RWopsHandle, *const c_void, isize, isize)->isize,
    close: extern fn (RWopsHandle)->c_int
}
```

16a ⟨*SDL:* SDL *Methods* 7c⟩+≡ (7a) ◁10d 16e▷

```
pub fn load_bytes<'sdl>(&'sdl self, bytes: &'static [u8])
->Result<RWops<'sdl>, Box<dyn Error>>
{ Ok( unsafe{ SDL_RWFromConstMem(bytes.as_ptr(),
                                 bytes.len() as c_int).unwrap()? } ) }
```

2.11 *PNG loading (and Surfaces)*

16b ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁15d 17a▷

```
#[repr(transparent)] #[derive(Debug,Copy,Clone)]
pub struct SurfaceHandle(*const c_void);
#[repr(transparent)] #[derive(Debug)]
pub struct Surface<'sdl>(SurfaceHandle, PhantomData<&'sdl SDL>);
impl<'sdl> Drop for Surface<'sdl> { fn drop(&mut self) { unsafe {
    SDL_FreeSurface(self.0); }}}
impl<'sdl> IsErr for Surface<'sdl> { fn is_err(&self)->bool {
    (self.0).0.is_null() }}
```

16c ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁15c 16f▷

```
fn SDL_FreeSurface(h:SurfaceHandle);
```

To load the images, we'll use a helper function out of `libSDL2_image`.

16d ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁14a 18a▷

```
#[link(name="SDL2_image")] extern
{   fn IMG_LoadPNG_RW<'a>(h:RWopsHandle)->ErrCheck<Surface<'a>>; }
```

16e ⟨*SDL:* SDL *Methods* 7c⟩+≡ (7a) ◁16a 19b▷

```
pub fn load_png<'sdl>(&'sdl self, bytes: &'static [u8])
->Result<Surface<'sdl>, Box<dyn Error>>
{ Ok( unsafe { IMG_LoadPNG_RW(self.load_bytes(bytes)?.0)
               .unwrap()?                               })}
```

2.12 *Texture loading*

Now that we have an image loaded into CPU memory, we need to send it to the GPU so that it can be rendered. This will be a small game, so there's no need to worry about running out of graphics memory (I hope).

16f ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁16c 17d▷

```
fn SDL_CreateTextureFromSurface<'r>
( r:RendererHandle, s:SurfaceHandle ) -> ErrCheck<Texture<'r>>;
fn SDL_DestroyTexture(tex: TextureHandle);
```

17a ⟨*SDL: C Types* 5c⟩+≡ (5a) ◁16b

```
#[repr(transparent)] #[derive(Debug,Copy,Clone)]
pub struct TextureHandle(*const c_void);
#[repr(transparent)] #[derive(Debug)]
pub struct Texture<'r>(TextureHandle, PhantomData<&'r Renderer<'r>>);
impl<'r> Drop for Texture<'r> { fn drop(&mut self) { unsafe {
    SDL_DestroyTexture(self.0); }}}
impl<'sdl> IsErr for Texture<'sdl> { fn is_err(&self)->bool {
    (self.0).0.is_null() }}
```

17b ⟨*SDL:* Renderer *methods* 14b⟩+≡ (13a) ◁14b

```
pub fn load_texture<'r>(&'r self, s: &Surface<'r>)
->Result<Texture<'r>, Box<dyn Error>>
{ Ok( unsafe { SDL_CreateTextureFromSurface(self.0, s.0)
              .unwrap()?                                })}
```

2.13 *Asset Organization*

Since we're going to have a bunch of different assets, it probably make sense to simplify the process of loading them a bit.

17c ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁13c 20▷

```
macro_rules!
PNG { ( $file:tt) => { renderer.load_texture(
                           &sdl.load_png(include_bytes!(
                                                   $file))?)? }};
```

2.14 *Texture rendering*

The texture is fully loaded now, so we can find out information like its size and then use that to render it to the screen.

17d ⟨*SDL: C Functions* 5b⟩+≡ (5a) ◁16f

```
fn SDL_QueryTexture(tex:    TextureHandle,
                    format: *mut u32,
                    access: *mut c_int,
                    w:      *mut c_int,
                    h:      *mut c_int) -> ErrCheck<c_int>;
fn SDL_RenderCopy(r:   RendererHandle,
                  t:   TextureHandle,
                  src: *const Rect,
                  dst: *const Rect    ) -> ErrCheck<c_int>;
```

18a ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁ 16d 19a ▷

```
impl<'r> Texture<'r> {
    pub fn get_size(&self) -> Result<(c_int, c_int), Box<dyn Error>>
    {   let mut w:c_int = 0;
        let mut h:c_int = 0;
        unsafe { SDL_QueryTexture(self.0,
                                  null_mut(),
                                  null_mut(),
                                  &mut w as *mut _,
                                  &mut h as *mut _) }.unwrap()?;
        Ok((w,h)) }}
```

18b ⟨*SDL:* Canvas *methods* 15a⟩+≡ (14a) ◁ 15a

```
pub fn blit(&self, tex:&Texture, dest:&Rect)
-> Result<(), Box<dyn Error>>
{ Ok( unsafe { SDL_RenderCopy((self.0).0,
                              tex.0,
                              null(),
                             dest as *const _).unwrap()?; })}
pub fn blit_center(&self, tex:&Texture, pos:(i32,i32))
-> Result<(), Box<dyn Error>>
{ let (w,h) = tex.get_size()?;
  let dest = Rect { x: pos.0 - w/2, y: pos.1 - h/2, w, h };
  Ok(self.blit(tex, &dest)?)                               }
```

2.15 *Font Loading*

In order to minimize the amount of graphics that need to be drawn by hand, I want to be able to use text strings programatically. For this, I'm going to use `libSDL2_ttf`.

19a ⟨*SDL: Definitions* 5a⟩+≡ (4a) ◁ 18a

```
#[link(name="SDL2_ttf")] extern {
    fn TTF_Init()->c_int;
    fn TTF_OpenFontRW<'sdl>(src   : RWopsHandle,
                            free  : c_int,
                            ptsize: c_int        )->Font<'sdl>;
    fn TTF_CloseFont(f: FontHandle);
    fn TTF_SetFontStyle(f: FontHandle, style:c_int);
    fn TTF_RenderUTF8_Blended<'sdl>(f   : FontHandle,
                                    text: *const c_char,
                                    fg  : Color         )
                                       ->Surface<'sdl>;
}

#[repr(C)] #[derive(Debug,Copy,Clone)]
pub struct Color { pub r: u8, pub g: u8, pub b:u8, pub  a:u8 }
#[repr(transparent)] #[derive(Debug,Copy,Clone)]
struct FontHandle(*const c_void);
#[repr(transparent)] #[derive(Debug)]
pub struct Font<'sdl>(FontHandle, PhantomData<&'sdl SDL>);
impl<'sdl> Drop for Font<'sdl> { fn drop(&mut self) { unsafe {
    TTF_CloseFont(self.0); }}}
impl<'sdl> Font<'sdl> {
    pub fn bold(self)->Self { unsafe { TTF_SetFontStyle(self.0,1);}; self}
    pub fn render(&self, text: &str, color: Color)
    ->Result<Surface<'sdl>, Box<dyn Error>> {
        let c_text = CString::new(text)?;
        let surf = unsafe { TTF_RenderUTF8_Blended(self.0,
                                                   c_text.as_ptr(),
                                                    color) };
        assert!(!(surf.0).0.is_null());
        Ok(surf)     }}
```

19b ⟨*SDL:* SDL *Methods* 7c⟩+≡ (7a) ◁ 16e

```
pub fn load_font<'sdl>(&'sdl self, size: c_int, bytes: &'static [u8])
->Result<Font<'sdl>, Box<dyn Error>>
{    static SDLTTF_INIT:Once = Once::new();
    SDLTTF_INIT.call_once(|| { unsafe { TTF_Init(); } });
    Ok(unsafe { TTF_OpenFontRW( self.load_bytes(bytes)?.0,
                                0, size                   )})}
```

20 ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁ 17c 30b ▷

```
let font = sdl.load_font(36, include_bytes!("Go-Mono.ttf"))?;
let boldfont = sdl.load_font(36, include_bytes!("Go-Mono.ttf"))?.bold();
macro_rules!
DRAW_TEXT { ( $str:expr, $color:expr) =>
        { renderer.load_texture(
                    &font.render($str, { let (r,g,b,a)=$color;
                                         Color { r,g,b,a }  })?)?}};
macro_rules!
DRAW_BOLD { ( $str:expr, $color:expr) =>
        { renderer.load_texture(
                    &boldfont.render($str, { let (r,g,b,a)=$color;
                                             Color { r,g,b,a }  })?)?}};
```

3 *Async & Futures*

In what was almost certainly a huge waste of time from a competition perspective, I spent several hours figuring out how Rust's async system works. The plan is to let me write functions that alter the gamestate over longer timescales than a frame.

3.1 *Executor*

Before it can do anything, we need to write a scheduler, which is responsible for calling `Future::poll()` at the top level to advance the program.

The `PendingTask` structure represents a top-level task that's not currently blocked.

21a ⟨*FUT: Definitions* 21a⟩≡ (4a) 21b ▷

```
use std::cell::*;
use std::rc::Rc;
use std::pin::*;
use std::task::*;
use std::future::Future;
use std::fmt::Debug;

type PendingTaskInner =
    RefCell<Option<Pin<Box<dyn Future<Output = ()> + 'static>>>>;

#[derive(Clone)]
pub struct PendingTask(Rc<PendingTaskInner>);
impl PendingTask { ⟨FUT: PendingTask methods 23a⟩ }
```

For the queue of pending tasks, we can use a static `Vec`:

21b ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 21a 22a ▷

```
fn task_queue() -> RefMut<'static, Vec<PendingTask>> {
    static mut QUEUE: Option<RefCell<Vec<PendingTask>>> = None;
    static INIT: std::sync::Once = std::sync::Once::new();
    INIT.call_once(|| { unsafe { QUEUE = Some(RefCell::new(vec![])); }});
    unsafe { QUEUE.as_ref().unwrap().borrow_mut() }
}
```

The enqueue function is the primary entry point for adding new tasks to the queue. It can be called as `enqueue(async { ...; });`. The newly-created PendingTask is returned so that the caller can monitor the status, but there is no need to retain it— the task will remain scheduled even if the caller drops this return result.

22a ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 21b 22b ▷

```
pub fn enqueue(f: impl Future<Output = ()> + 'static)->PendingTask
{   let result = PendingTask(
                    Rc::new( RefCell::new( Some( Box::pin(f)))) );
    task_queue().push( result.clone() );
    result                                                        }
```

Its counterpart is run_tasks, which will advance all tasks as far a possible, and return as soon as they are all blocked. For our purposes, this will be an opportunity to render a frame and check for new input.

22b ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 22a 23b ▷

```
pub fn run_tasks() { while run_one_task() {} }
fn run_one_task()->bool
{   let task = task_queue().pop();
    match task { None    =>   false,
                 Some(t) => { ⟨FUT: Run Task t 22c⟩
                              true                    }}}
```

To actually run a task, the executor needs to construct a Waker to pass as the argument to Future.poll. The waker stores a single data pointer, which is raw and outside of the compiler's reference checking systems.

22c ⟨*FUT: Run Task* t 22c⟩≡ (22b)

```
let w = unsafe { Waker::from_raw( RawWaker::new(t.clone_as_ptr(),
                                                &VTABLE        ) )};
let mut optfut = t.0.borrow_mut();
if optfut.is_some()
{   let result = optfut.as_mut().unwrap()
                 .as_mut().poll(&mut Context::from_waker(&w));
    if result.is_ready() { *optfut = None; }                  };
```

In order to round-trip through the raw pointer API exposed by Rust, we need to define to- and from-pointer methods for our pending tasks. This is the primary reason they're wrapped in an `Rc`— I wanted to use its `from_raw` and `into_raw` methods to do the heavy lifting.

23a ⟨*FUT:* `PendingTask` *methods* 23a⟩≡ (21a)

```
unsafe fn from_ptr(p: *const ())->PendingTask {
    PendingTask(Rc::from_raw(p as *const PendingTaskInner))
}
fn clone_as_ptr(&self)->*const () {
    Rc::into_raw(Rc::clone(&self.0)) as *const ()
}
```

We also need to provide a static vtable of method implementations for the `Waker`. They're mostly straightforward from the descriptions in the documentation; the only thing to be careful of is paying attention to which should deallocate the stored data and which shouldn't.

23b ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁22b 23c▷

```
static VTABLE:RawWakerVTable = RawWakerVTable::new(
    raw_clone, raw_wake, raw_wake_by_ref, raw_drop
);
```

`raw_clone` is called to duplicate a `RawWaker`. We reconstruct the pending task, clone it, and then convert the original task back to a raw pointer so that it doesn't get cleaned up at the end of the function.

23c ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁23b 23d▷

```
unsafe fn raw_clone(p: *const ()) -> RawWaker {
    let t = PendingTask::from_ptr(p);
    let result = RawWaker::new(t.clone_as_ptr(), &VTABLE);
    assert_eq!(Rc::into_raw(t.0) as *const (), p);
    result
}
```

`raw_wake` reconstructs the task and pushes it onto the task queue to be run at the next opportunity.

23d ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁23c 24a▷

```
unsafe fn raw_wake(p: *const ()) {
    let t = PendingTask::from_ptr(p);
    task_queue().push(t);
}
```

raw_wake_by_ref is a combination of the previous two. It pushes a clone of the reconstructed task onto the queue, and then converts the original back to a raw pointer so that it doesn't get dropped.

24a ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 23d 24b ▷

```
unsafe fn raw_wake_by_ref(p: *const ()) {
    let t = PendingTask::from_ptr(p);
    task_queue().push(t.clone());
    assert_eq!(Rc::into_raw(t.0) as *const (), p);
}
```

raw_drop is the simplest of all. It reconstructs the pending task and then does nothing else. When it leaves scope at the bottom of this function, the normal Drop machinery will take care of whatever needs to be done.

24b ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 24a 24c ▷

```
unsafe fn raw_drop(p: *const ()) {
    let _t = PendingTask::from_ptr(p);
}
```

3.2 Notification Futures

I plan to make only small pieces of the game code async, like sounds and animation. Most of the logic will be running on a traditional every-frame cadence, so I'm going to need a way to inject events from the traditional game loop into the async system. That means writing my own Future.

NotificationPool will handle dispatching a single value from the main loop to futures that are waiting for it.

Thoughts from the next morning: This all works fine, but there is probably some kind of memory leak in here. At the very least, futures will keep the notification pool alive even after anyone who might use it to post a notification is gone. If a notification has been posted, this is essentially a shared reference to the result, which is fine. If it hasn't, though, the pool holds a bunch of thread wakers that will never get called and hold references back into the pool.

24c ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 24b 25d ▷

```
use std::sync::{Arc,Mutex};
#[derive(Debug)]
pub struct NotificationPool<T:Clone+Debug>(Mutex<NotifPoolData<T>>);
#[derive(Debug)]
struct NotifPoolData<T> { result: Option<T>,
                          wakers: Vec<Option<Waker>> }
impl<T:Clone+Debug> NotificationPool<T>
{ ⟨FUT: NotificationPool methods 25a⟩ }
```

In general, these will be held in an Arc so that their data can stick around as long as there are potential readers for it

25a ⟨*FUT:* NotificationPool *methods* 25a⟩≡ (24c) 25b ▷

```
pub fn new()->Arc<Self> {
    Arc::new(
        NotificationPool(
            Mutex::new( NotifPoolData{ result:None,
                                       wakers:vec![] })))}
```

The core functionality is notify, which sends its argument to all current and future tasks that might need the value. It's important that we release our self lock before attempting to run the Wakers, as that may cause code to run (via Drop methods) that attempts to obtain the lock.

25b ⟨*FUT:* NotificationPool *methods* 25a⟩+≡ (24c) ◁ 25a 25c ▷

```
pub fn notify(&self, result:T) {
    let wakers =
    {   let mut self_mut = self.0.lock().unwrap();
        assert!(self_mut.result.is_none());
        self_mut.result = Some(result);
        std::mem::replace(&mut self_mut.wakers, vec![]) };
    for maybe_w in wakers.into_iter()
    {   if let Some(waker) = maybe_w { waker.wake(); }}     }
```

And its corollary, wait, that returns a future which will eventually yield the notification.

25c ⟨*FUT:* NotificationPool *methods* 25a⟩+≡ (24c) ◁ 25b

```
pub fn wait(self: Arc<Self>)->NotificationFuture<T> {
    let id: usize =
    {   let mut self_mut = self.0.lock().unwrap();
        self_mut.wakers.push(None);
        self_mut.wakers.len() - 1                 };
    NotificationFuture { id, pool: self }             }
```

25d ⟨*FUT:* *Definitions* 21a⟩+≡ (4a) ◁ 24c 26a ▷

```
#[derive(Debug)]
pub struct NotificationFuture<T:Clone+Debug>
{ pool: Arc<NotificationPool<T>>,
  id  : usize                    }
```

Implementing the `Future` trait means we have to do one of two things when `poll` is called: If the result is ready, we return it. But if it isn't, we need to store a copy of the given `Waker` such that it gets woken up when the result becomes ready.

26a ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 25d 26b ▷

```
impl<T:Clone+Debug> Future for NotificationFuture<T>
{   type Output = T;
    fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<T>
    {   let mut pool = self.pool.0.lock().unwrap();
        if let Some(ref result) = pool.result
        {   Poll::Ready(result.clone()) }
        else {  pool.wakers[self.id] = Some( cx.waker()
                                               .clone() );
                Poll::Pending                                } } }
```

Since the `NotificationFuture` is responsible for managing its waker in the `NotificationPool`, we should implement the `Drop` trait to clean that up as well. Because this requires taking the pool lock, we need to be careful about what gets dropped while the lock is held.

26b ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 26a 26c ▷

```
impl<T:Clone+Debug> Drop for NotificationFuture<T>
{ fn drop(&mut self)
  { let _w: Option<Waker> // Dont't drop while holding lock
        = { let mut pool = self.pool.0.lock().unwrap();
            if pool.wakers.len() > self.id
            { std::mem::replace(&mut pool.wakers[self.id],
                                None                        )}
            else { None }                                    };}}
```

As this future is essentially just an `Arc` reference to the notification pool, we can easily make a reasonable `Clone` implementation as well. I have no idea if it'll actuall be useful, as futures in general aren't cloneable.

26c ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 26b 27a ▷

```
impl<T:Clone+Debug> Clone for NotificationFuture<T>
{   fn clone(&self)->NotificationFuture<T>
    {   Arc::clone(&self.pool).wait()    }  }
```

3.3 Notification Streams

While sending a single value into the futures system is a good start, it's not really as expressive as I would like. If that's a compound value, however, we can do more interesting things with it. In particular, I want to be sending `Result<(T, Future<...>), Err>`. This will let me send and endless stream of values, and also throw errors in the receiving threads. Unfortunately, error types aren't guaranteed to be cloneable and I don't have time to be fighting with the type checker right now.

27a ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 26c 28a ▷

```
#[derive(Clone,Debug)]
struct StreamArg<T:Clone+Debug+'static>( T,
                                NotificationFuture<StreamArg<T>>);
pub struct StreamSource<T:Clone+Debug+'static>
{    pool: Arc<NotificationPool<StreamArg<T>>>  }
impl<T:Clone+Debug> StreamSource<T> { ⟨FUT: StreamSource methods 27b⟩ }
```

Creating a new stream only requires making a new notification pool:

27b ⟨*FUT:* StreamSource *methods* 27b⟩≡ (27a) 27c ▷

```
pub fn new() -> StreamSource<T>
{   StreamSource { pool: NotificationPool::new() } }
```

To send the next value, we make a new notification pool, and send the future for that pool along with the sent value. The old pool gets orphaned, and the reference counter will clean it up once there are no readers left that need its result.

27c ⟨*FUT:* StreamSource *methods* 27b⟩+≡ (27a) ◁ 27b 27d ▷

```
pub fn send(&mut self, msg:T)
{   let next_pool = NotificationPool::<StreamArg<T>>::new();
    let payload = StreamArg( msg, Arc::clone(&next_pool).wait());
    self.pool.notify(payload);
    self.pool = next_pool;                                    }
```

To get the values back out, use `watch` to create a `StreamReader`.

27d ⟨*FUT:* StreamSource *methods* 27b⟩+≡ (27a) ◁ 27c

```
pub fn watch(&self) -> StreamReader<T>
{   StreamReader { fut: Arc::clone(&self.pool).wait() } }
```

The `StreamReader` doesn't need to know anything except for the future that the `StreamSource` will be populating. Because this kind of future is cloneable, the stream is also cloneable. Cloning the stream will produce another stream that starts at the same point as when the original was cloned.

28a ⟨*FUT: Definitions* 21a⟩+≡ (4a) ◁ 27a

```
#[derive(Clone, Debug)]
pub struct StreamReader<T:Clone+Debug+'static>
{   fut: NotificationFuture<StreamArg<T>>   }

impl<T:Clone+Debug+'static> StreamReader<T>
{ ⟨FUT: StreamReader methods 28b⟩ }
```

The primary method is `next()`. Awaiting this will produce a value that was provided to the source through `send()`, and the new future that represents the remaining stream values is swapped into the structure so that subsequent calls will get the newer values.

28b ⟨*FUT:* StreamReader *methods* 28b⟩≡ (28a) 28c ▷

```
pub async fn next(&mut self) -> T
{   let StreamArg(result, new_fut) = self.fut.clone().await;
    self.fut = new_fut;
    return result;                                       }
```

As a convenience, we'll also define a `filter` method that can produce a stream with certain elements removed:

28c ⟨*FUT:* StreamReader *methods* 28b⟩+≡ (28a) ◁ 28b 29a ▷

```
pub fn filter(self, f: impl Fn(&T)->bool + 'static) -> StreamReader<T>
{   let mut source = self;
    let mut newsource = StreamSource::<T>::new();
    let result = newsource.watch();
    enqueue(async move
    {   loop {  let val:T = source.next().await;
                if f(&val) { newsource.send(val);   }}});
    result                                               }
```

And a `map` method to transform the contained datatype:

29a ⟨*FUT:* StreamReader *methods* 28b⟩+≡ (28a) ◁ 28c 29b ▷

```
pub fn map<R>(self, f: impl Fn(T)->R + 'static) -> StreamReader<R>
where R: Debug+Clone+'static
{   let mut source = self;
    let mut newsource = StreamSource::<R>::new();
    let result = newsource.watch();
    enqueue(async move
    {   loop {  let val:T = source.next().await;
                newsource.send(f(val));            }});
    result                                                }
```

29b ⟨*FUT:* StreamReader *methods* 28b⟩+≡ (28a) ◁ 29a

```
pub fn filtermap<R>(self, f: impl Fn(T)->Option<R> + 'static) -> StreamReader<R>
where R: Debug+Clone+'static
{   let mut source = self;
    let mut newsource = StreamSource::<R>::new();
    let result = newsource.watch();
    enqueue(async move
    {   loop {  let val:T = source.next().await;
                if let Some(r) = f(val) { newsource.send(r) }}});
    result                                                }
```

4 *Putting it all together*

We now have all the building blocks we need to put a game together. The bulk of the game logic will be async code in the `sim` module and have limited interaction with the underlying frame-by-frame activities. Input to the game logic comes from a single stream that reflects all of the events that happen, including frame timing information.

30a ⟨*SIM: Definitions* 30a⟩≡ (4a) 31a ▷

```
#[derive(Debug, Copy, Clone, Eq, PartialEq, Ord, PartialOrd)]
pub enum Command { ⟨SIM: Command variants 30c⟩ }
```

30b ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁ 20 34c ▷

```
let mut command_queue = StreamSource::<sim::Command>::new();
let command_stream = command_queue.watch();
enqueue(sim::start(command_stream));
```

`run_tasks()` returns when there is no more progress to be made, even if the tasks haven't completed. This means that we can have tasks that are infinite loops, and they'll get paused once they require information that doesn't exist yet. At that point, we continue the main game loop: render the updated game state to the screen, collect more user input, and then resume the game logic.

30c ⟨*SIM:* `Command` *variants* 30c⟩≡ (30a) 35f ▷

```
FrameAdvance( u32 ),
```

30d ⟨*MAIN: Run Simulation* 30d⟩≡ (4d)

```
command_queue.send(sim::Command::FrameAdvance(sdl.ticks()));
run_tasks();
```

4.1 Game state management

The previous section discussed how we provide new information to the game logic, but we still need to get information out of it. Out of expediency, I have a single mutable global variable that contains the entire game state, protected by a `RefCell`. This is not a great solution because it requires careful programming to avoid game-crashing bugs, particularly never holding the state's reference across an `await`. Better would be some kind of hierarchical system, where each async thread can effectively own a small piece of the game state and periodically publish updates to the world.

31a ⟨*SIM: Definitions* 30a⟩+≡ (4a) ◁ 30a 31b ▷

```
use crate::fut::*;
use std::cell::RefCell;

#[derive(Debug, Default)]
pub struct GameState { ⟨SIM: GameState fields 34a⟩ }
impl GameState
{ pub fn current() -> &'static RefCell<GameState>
  { static mut STATE: Option<RefCell<GameState>> = None;
    static INIT: std::sync::Once = std::sync::Once::new();
    INIT.call_once(
            || { unsafe
                 { STATE = Some(RefCell::new(
                                   Default::default())); } } );
    unsafe { STATE.as_ref().unwrap() }                          }}
```

This macro is just some syntactic sugar on top of `filtermap` for streams:

31b ⟨*SIM: Definitions* 30a⟩+≡ (4a) ◁ 31a 32a ▷

```
macro_rules! select
{ ($s:ident : $p:pat => $e:expr) => { select!(($s): $p => $e) };
  (($stream:expr): $pat:pat => $expr:expr) =>
  { ($stream).clone().filtermap(|value|
    { if let $pat = value { Some($expr) } else { None } }) }; }
```

The `start` method initializes the game state, spawns various worker tasks, and then exits.

32a ⟨*SIM: Definitions* 30a⟩+≡ (4a) ◁31b 34b▷

```
pub async fn start(firehose: StreamReader<Command>) {
    use Command::*;
    {   let mut gs = GameState::current().borrow_mut();
        ⟨SIM: Initialize gs:GameState 34e⟩
    }

    use std::sync::atomic::*;
    use std::sync::atomic::Ordering::*;
    static T_PREV:AtomicU32 = AtomicU32::new(0);
    let timing = select!( firehose: FrameAdvance(t)
                          => (t, t-T_PREV.swap(t, Relaxed)) );
    ⟨SIM: Start Workers 36b⟩
}
```

4.2 Render Order

So far, everything has pretty much been put together in the source code file in the order I came up with it. This works fine for things liks function definitions, but for some things, the ordering is quite important. When rendering a frame, for instance, the code order is effectively the same as the stacking order of the things that are being drawn: items later in the list can overdraw things earlier on the list.

Because of this, I have consolidated the overall drawing order here. Each individual drawing task is enclosed in its own block so that defined variables don't leak out to the others.

32b ⟨*MAIN: Draw Frame* 32b⟩≡ (4d)

```
{ let game_state = sim::GameState::current().borrow();
  let canvas = renderer.start_frame((128,0,255))?;
  { ⟨MAIN: Draw Terrain 34d⟩ }
  { ⟨MAIN: Draw Floor Markings 37e⟩ }
  { ⟨MAIN: Draw Player 35e⟩ }
}
```

4.3 *Coordinates*

The gameplay happens on a hex grid. I'm using a rectangular coordinate system, where only half of the coordinates are valid spaces, in a checkerboard pattern.

I have a couple of helper functions to translate between world space and screen space:

33 ⟨*MAIN: Definitions* 4d⟩+≡ (4a) ◁ 4d

```
fn grid_to_px(obj:(f32, f32), cam:(f32, f32))->(i32,i32)
{ (((obj.0-cam.0)*36.).floor() as i32+400,
   ((obj.1-cam.1)*21.).floor() as i32+300) }
fn igrid_to_px(obj:(usize, usize), cam:(f32, f32))->(i32,i32)
{ grid_to_px((obj.0 as f32, obj.1 as f32), cam) }
```

5 *Game Elements*

Here you will find descriptions of all the game elements and logic.

5.1 *Terrain*

The terrain is stored in a two dimensional vector, with the y coordinate corresponding to the outermost vector.

34a ⟨*SIM: GameState fields* 34a⟩≡ (31a) 35b ▷

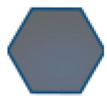
```
pub terrain:Vec<Vec<Terrain>>,
```

The actual tile definition is an `enum` which contains two variants, one for empty spaces and another for walkable tiles.

34b ⟨*SIM: Definitions* 30a⟩+≡ (4a) ◁ 32a 35a ▷

```
#[derive(Debug,Eq,PartialEq)]
pub enum Terrain { Empty, Floor }
```

Tiles are drawn from a PNG source, as shown to the right.



Flathex.png

34c ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁ 30b 35d ▷

```
let tex_floor = PNG!("flathex.png");
```

Because the terrain is stored in row-major order, it will be drawn top-down on the screen, so lower rows can draw over higher rows. This would allow for a 2.5D rendering scheme in the future.

34d ⟨*MAIN: Draw Terrain* 34d⟩≡ (32b)

```
for y in 0..game_state.terrain.len()
{ for x in 0..game_state.terrain[y].len()
  { if game_state.terrain[y][x] == sim::Terrain::Floor
    { let center = igrid_to_px((x, y), game_state.camera);
      canvas.blit_center(&tex_floor, center)?;             } } }
```

When setting up a new game, the terrain gets filled such that the black squares of a checkerboard have floor and the red squares are empty.

34e ⟨*SIM: Initialize* gs:GameState 34e⟩≡ (32a) 35c ▷

```
for y in 0..40
{   gs.terrain.push(vec![]);
    for x in 0..40
    {   gs.terrain[y].push( match (x^y)&1
                               { 0 => Terrain::Floor,
                                 _ => Terrain::Empty } ); } }
```


5.2 *Player*

The player gets a dedicated `struct` in the game state, which currently only contains their position, but could easily be expanded later to accomidate future gameplay changes.

35a ⟨*SIM: Definitions* 30a⟩+≡ (4a) ◁34b 36c▷

```
#[derive(Debug,Default)]
pub struct Player { pub loc: (usize, usize) }
```

35b ⟨*SIM:* `GameState` *fields* 34a⟩+≡ (31a) ◁34a 36d▷

```
pub player:Player,
```

The starting location of the player is hardcoded to be the center of the playfield.

35c ⟨*SIM: Initialize* `gs:GameState` 34e⟩+≡ (32a) ◁34e 36e▷

```
gs.player.loc = (20,20);
```

In an homage to the text-based rougelikes (and a bid to avoid making real artwork), the player is represented by a blue `@` sign. I couldn't find any shade that had enough contrast with the slightly blue-tinted ground tiles, so I simulate a dark outline by drawing a boldfaced copy first, with a normal-weight version on top.

35d ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁34c 38c▷

```
let tex_player    = DRAW_TEXT!("@", (88,100,232,255));
let tex_player_bg = DRAW_BOLD!("@", (22,25,60,255));
```

35e ⟨*MAIN: Draw Player* 35e⟩≡ (32b) 38d▷

```
let loc = igrid_to_px(game_state.player.loc, game_state.camera);
canvas.blit_center(&tex_player_bg, loc)?;
canvas.blit_center(&tex_player,    loc)?;
```

MOVEMENT IS CONTROLLED by the keyboard keys q,w,e,a,s, and d. We use a `Step` command to let the runtime send these to us, and make the corresponding `KeyDn` events inject them into the command stream.

35f ⟨*SIM:* `Command` *variants* 30c⟩+≡ (30a) ◁30c

```
Step( i32, i32 ),
```

36a ⟨*MAIN: Event Handlers* 11c⟩+≡ (10e) ◁ 11c

```
SdlEvent::KeyDn(0x014, _) => { command_queue.send(Step(-1,-1)); }
SdlEvent::KeyDn(0x01a, _) => { command_queue.send(Step( 0,-2)); }
SdlEvent::KeyDn(0x008, _) => { command_queue.send(Step( 1,-1)); }
SdlEvent::KeyDn(0x004, _) => { command_queue.send(Step(-1, 1)); }
SdlEvent::KeyDn(0x016, _) => { command_queue.send(Step( 0, 2)); }
SdlEvent::KeyDn(0x007, _) => { command_queue.send(Step( 1, 1)); }
SdlEvent::KeyUp(_,_) => {}
SdlEvent::KeyDn(_,_) => {}
```

We also have a worker that handles the step requests, and moves the player if the requested space is clear.

36b ⟨*SIM: Start Workers* 36b⟩≡ (32a) 37a ▷

```
let steps = select!(firehose: Step(x, y) => (x, y));
enqueue(stepper(steps));
```

36c ⟨*SIM: Definitions* 30a⟩+≡ (4a) ◁ 35a 37b ▷

```
async fn stepper(mut steps: StreamReader<(i32,i32)>) { loop
{   let (dx, dy) = steps.next().await;
    let mut gs = GameState::current().borrow_mut();
    let (x,y) = (gs.player.loc.0 as i32 + dx,
                 gs.player.loc.1 as i32 + dy);
    if x < 0 || y < 0 { continue }
    let loc:(usize, usize) = (x as usize, y as usize);
    if loc.1 >= gs.terrain.len()        { continue };
    if loc.0 >= gs.terrain[loc.1].len() { continue };
    if gs.wrecks.contains(&loc)         { continue };
    match gs.terrain[loc.1][loc.0]
    {   Terrain::Floor => { gs.player.loc = loc; }
        Terrain::Empty => { continue }             };
    ⟨SIM: Enemy Turn 38e⟩
}}
```

5.3 Camera

The game uses an ad-hoc orthographic renderer, so there isn't much for a camera to do. It is represented by a tuple in the game state that is the world position of the center of the screen.

36d ⟨*SIM:* `GameState` *fields* 34a⟩+≡ (31a) ◁ 35b 37c ▷

```
pub camera:(f32, f32),
```

36e ⟨*SIM: Initialize* `gs:GameState` 34e⟩+≡ (32a) ◁ 35c 37d ▷

```
gs.camera = (15.,15.);
```

It should generally follow the player around. If it made the same jerky, jumping movements that the player did, things would quickly get disorienting. Using an exponential approach is convenient, as the calculation is can rely on just the difference between the current and desired positions, and the size of the time step. Every frame, I multiply the distance between the camera and the player by $0.9999^{\delta t}$, which seems to be a reasonable speed.

37a ⟨*SIM: Start Workers* 36b⟩+≡ (32a) ◁36b

```
enqueue(camera_track(timing.clone()));
```

37b ⟨*SIM: Definitions* 30a⟩+≡ (4a) ◁36c 41▷

```
async fn camera_track(mut times: StreamReader<(u32,u32)>) { loop
{   let (_t, dt) = times.next().await;
    let mut gs = GameState::current().borrow_mut();
    let coeff: f32 = 0.9999f32.powi(dt as i32);
    gs.camera.0 = gs.player.loc.0 as f32
                + coeff * (gs.camera.0 - gs.player.loc.0 as f32);
    gs.camera.1 = gs.player.loc.1 as f32
                + coeff * (gs.camera.1 - gs.player.loc.1 as f32); }}
```

5.4 *Floor markings*

In addition to the tiles, the floor has some markings to teach the player how to move around and to provide a fixed point to help them stay oriented in the otherwise featureless plane the game is operating on at the moment.

37c ⟨*SIM:* `GameState` *fields* 34a⟩+≡ (31a) ◁36d 38a▷

```
pub floor_marks:Vec<(f32,f32,&'static str)>,
```

37d ⟨*SIM: Initialize* `gs:GameState` 34e⟩+≡ (32a) ◁36e 38b▷

```
for mark in [ (18.,18.,"q"), (20.,16.,"w"),
              (22.,18.,"e"), (18.,22.,"a"),
              (20.,24.,"s"), (22.,22.,"d") ].into_iter()
    { gs.floor_marks.push(*mark); }
```

I really should be caching the textures for these letters, but it was easier to rerender them every frame:

37e ⟨*MAIN: Draw Floor Markings* 37e⟩≡ (32b)

```
for (x,y,s) in game_state.floor_marks.iter() {
    canvas.blit_center(&DRAW_TEXT!(s,(88,88,111,55)),
                       grid_to_px((*x,*y), game_state.camera))?;}
```

5.5 *Enemies*

Much like the player, the enemies are rendered as a red X. As there can be several enemies active at a time, they are tracked in a `Vec` in the game state.

38a ⟨*SIM:* `GameState` *fields* 34a⟩+≡ (31a) ◁37c 40a▷

```
pub enemies:Vec<(usize, usize)>,
```

38b ⟨*SIM: Initialize* `gs:GameState` 34e⟩+≡ (32a) ◁37d 40b▷

```
gs.enemies = vec![(25,9), (3,3)];
```

38c ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁35d 40c▷

```
let tex_enemy    = DRAW_TEXT!("X", (232,100,88,255));
let tex_enemy_bg = DRAW_BOLD!("X", (60,25,22,255));
```

38d ⟨*MAIN: Draw Player* 35e⟩+≡ (32b) ◁35e 40d▷

```
for world_loc in game_state.enemies.iter()
{   let loc = igrid_to_px(*world_loc, game_state.camera);
    canvas.blit_center(&tex_enemy_bg, loc)?;
    canvas.blit_center(&tex_enemy,    loc)?;              }
```

The enemies all get a turn after any successful step by the player, so their movement logic also lives in `stepper`;

38e ⟨*SIM: Enemy Turn* 38e⟩≡ (36c)

```
for i in 0..gs.enemies.len() {⟨SIM: Enemy i gives chase 39a⟩}

let mut deadlist:Vec<usize> = vec![];
for i in 0..gs.enemies.len()
{   let mut dead:bool = false;
    { ⟨SIM: Enemy i checks for collisions 39b⟩ }
    if dead { deadlist.push(i) };
}
while !deadlist.is_empty()
{ let id = deadlist.pop().unwrap();
  println!("kill {:?}", id);
  gs.enemies.remove(id); }
if gs.enemies.len() < 2 {
    ⟨SIM: Spawn new enemies 39c⟩
}
```

The enemies always walk directly towards the player, as much as they are able.

39a ⟨*SIM: Enemy i gives chase 39a*⟩≡ (38e)

```
let dx = gs.player.loc.0 as i32 - gs.enemies[i].0 as i32;
let dy = gs.player.loc.1 as i32 - gs.enemies[i].1 as i32;
let mut e = &mut gs.enemies[i];
     if dx == 0 &&               dy >  0 {           e.1 += 2; }
else if dx <  0 && -3*dx < dy && dy >  0 {           e.1 += 2; }
else if dx >  0 &&  3*dx < dy && dy >  0 {           e.1 += 2; }
else if dx == 0 &&               dy <  0 {           e.1 -= 2; }
else if dx <  0 &&  3*dx > dy && dy <  0 {           e.1 -= 2; }
else if dx >  0 && -3*dx > dy && dy <  0 {           e.1 -= 2; }
else if dx >  0 &&                dy > 0 { e.0 += 1; e.1 += 1; }
else if dx >  0 &&                dy < 0 { e.0 += 1; e.1 -= 1; }
else if dx <  0 &&                dy < 0 { e.0 -= 1; e.1 -= 1; }
else if dx <  0 &&                dy > 0 { e.0 -= 1; e.1 += 1; }
```

39b ⟨*SIM: Enemy i checks for collisions 39b*⟩≡ (38e)

```
let loc = gs.enemies[i];
     if gs.player.loc == loc { return; } // Game Over
else if gs.wrecks.contains(&loc) { dead = true; }
else {
    for j in 0..gs.enemies.len() {
        if j != i && gs.enemies[j] == loc {
            dead = true; gs.wrecks.push(loc);
}}}
```

39c ⟨*SIM: Spawn new enemies 39c*⟩≡ (38e)

```
use std::sync::atomic::*;
static NUM_ENEMIES:AtomicUsize = AtomicUsize::new(3);
for n in 0..NUM_ENEMIES.fetch_add(1, Ordering::Relaxed) {
    loop {
        let coord = random_coord();
        if gs.wrecks.contains(&coord) {continue};
        if gs.enemies.contains(&coord) {continue};
        if (gs.player.loc.0 as i32 - coord.0 as i32).abs()
         + (gs.player.loc.1 as i32 - coord.1 as i32).abs()
         < 5 { continue };
        gs.enemies.push(coord);
        break;
    }
}
```

5.6 Wrecks

Wrecks occur when two enemies run into each other. They will destroy future enemies that run into them. They are tracked in a `Vec` in the game state.

40a ⟨*SIM:* `GameState` *fields* 34a⟩+≡ (31a) ◁ 38a

```
pub wrecks:Vec<(usize, usize)>,
```

40b ⟨*SIM: Initialize* `gs:GameState` 34e⟩+≡ (32a) ◁ 38b

40c ⟨*MAIN: Initialization* 7b⟩+≡ (4d) ◁ 38c

```
let tex_wreck = DRAW_BOLD!("#", (44,44,44,255));
```

40d ⟨*MAIN: Draw Player* 35e⟩+≡ (32b) ◁ 38d

```
for world_loc in game_state.wrecks.iter()
{   let loc = igrid_to_px(*world_loc, game_state.camera);
    canvas.blit_center(&tex_wreck,    loc)?;                }
```

6 Randomness

Rust doesn't have a random number generator in its standard library, but fortunately the only thing I need random samples of is grid coordinates. I used Python to generate a bunch of these:

41 ⟨*SIM: Definitions* 30a⟩+≡ (4a) ◁37b

```
static COORDS:[(usize,usize);200] = [(17, 35), (6, 22), (13, 31),
(28, 2), (15, 29), (34, 26), (3, 9), (14, 12), (32, 34), (5, 31),
(25, 37), (22, 28), (17, 31), (0, 22), (6, 32), (9, 1), (11, 39),
(36, 32), (1, 33), (33, 1), (5, 15), (32, 10), (36, 26), (15, 1),
(16, 12), (14, 28), (38, 38), (21, 25), (31, 11), (9, 23),
(0, 38), (22, 4), (36, 38), (13, 1), (34, 6), (22, 34), (24, 30),
(34, 2), (21, 19), (34, 34), (17, 11), (23, 23), (39, 29),
(13, 39), (23, 37), (25, 3), (27, 13), (34, 30), (0, 0), (5, 9),
(38, 18), (28, 6), (20, 14), (35, 37), (34, 30), (30, 8),
(16, 30), (34, 36), (25, 1), (19, 27), (32, 14), (24, 14),
(14, 6), (34, 20), (25, 23), (6, 30), (30, 34), (27, 27),
(17, 17), (16, 38), (5, 33), (4, 16), (8, 6), (38, 2), (34, 16),
(8, 14), (13, 39), (19, 21), (35, 21), (33, 15), (38, 4),
(22, 8), (36, 16), (18, 22), (1, 9), (16, 12), (11, 5), (23, 29),
(3, 37), (22, 34), (17, 37), (18, 8), (35, 19), (0, 34),
(39, 35), (27, 17), (31, 27), (36, 28), (26, 6), (30, 22),
(23, 19), (39, 11), (10, 24), (16, 14), (1, 35), (3, 37),
(10, 2), (5, 3), (6, 6), (28, 6), (31, 9), (24, 18), (33, 31),
(11, 13), (11, 17), (4, 36), (31, 13), (7, 37), (28, 10),
(34, 26), (3, 17), (12, 20), (33, 35), (27, 33), (9, 27),
(17, 11), (4, 20), (38, 32), (36, 4), (9, 3), (28, 38), (13, 35),
(28, 2), (25, 33), (12, 18), (16, 6), (37, 15), (7, 21),
(19, 15), (18, 8), (2, 8), (23, 1), (25, 9), (12, 8), (13, 39),
(21, 17), (23, 11), (35, 1), (1, 7), (1, 7), (5, 15), (1, 1),
(10, 14), (27, 13), (39, 17), (9, 33), (37, 19), (34, 32),
(12, 6), (32, 4), (23, 29), (13, 29), (37, 27), (2, 38), (9, 21),
(12, 38), (8, 6), (27, 7), (22, 2), (26, 32), (36, 28), (5, 27),
(36, 26), (22, 28), (20, 14), (17, 21), (29, 39), (15, 29),
(39, 21), (1, 13), (14, 16), (13, 9), (1, 37), (29, 23), (21, 1),
(35, 29), (18, 32), (37, 39), (24, 36), (7, 5), (12, 26),
(14, 26), (39, 29), (2, 4), (39, 31), (9, 39), (13, 5), (20, 38),
(33, 37), (9, 17)];
use std::sync::atomic::*;
static NEXT_COORD:AtomicUsize = AtomicUsize::new(0);
fn random_coord() -> (usize, usize) {
    let coord = NEXT_COORD.fetch_add(1, Ordering::Relaxed);
    return COORDS[coord % COORDS.len()];
}
```

7 Licensing

7.1 Licensing: Go Mono