Getting from C to D without Tripping

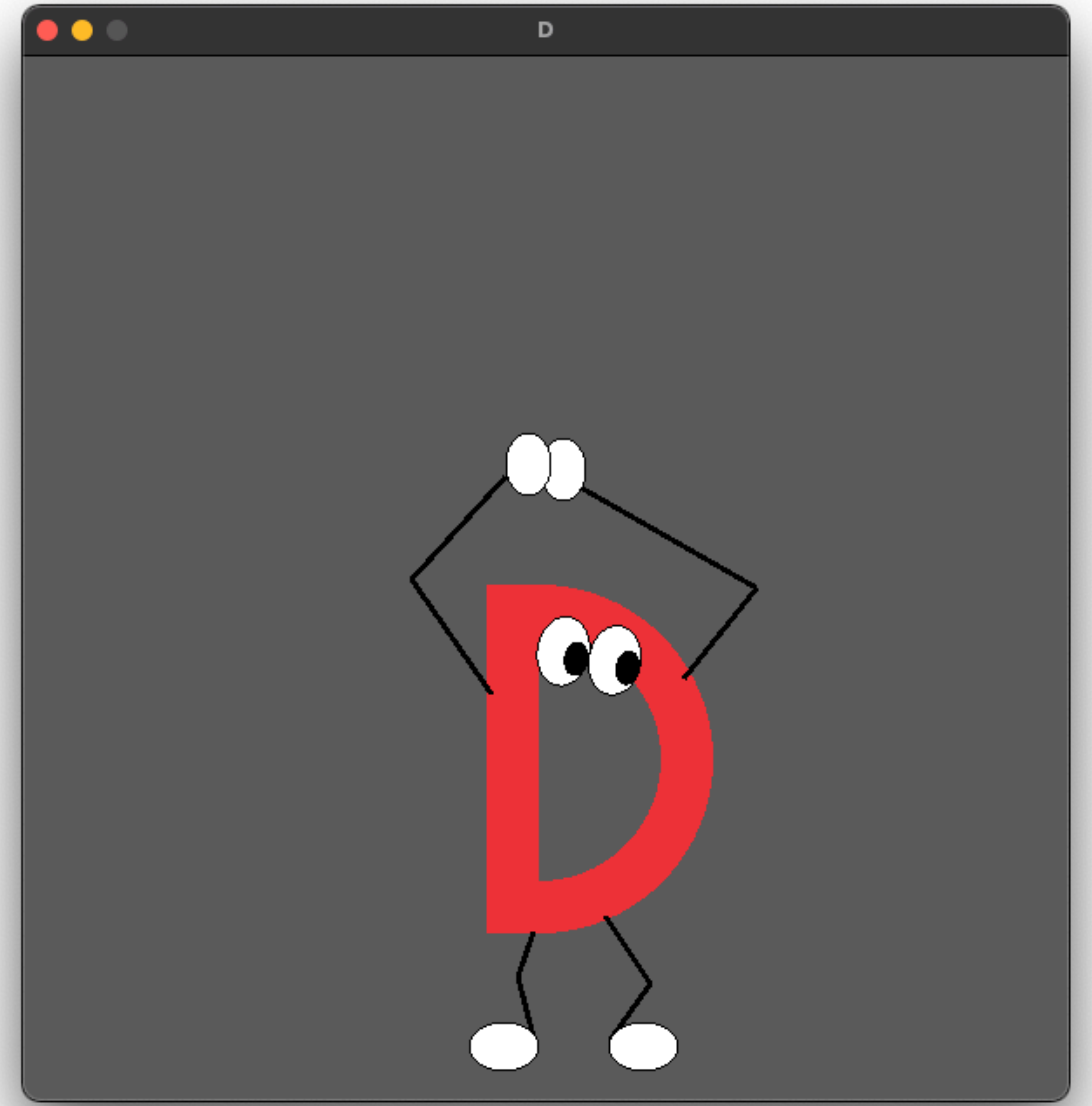
My adventures porting a large C library to D

Dconf 2023 London - Steven Schveighoffer

Why are we here?

How I face-planted many times so you don't have to

- History of how I became involved in this project
- Stage 1 - Bindings
- Stage 2 - Improvements to bindings
- Stage 3 - Porting
- Stage 4 - Making tools to help porting
- Stage 5 - Improve the API



A Bit of History

D as a teaching language

https://dlang.org/blog/2021/12/23/teaching-d-from-scratch-is-it-a-viable-first-language/

- Homeschool class on coding. What language to use?
- Not everyone had a full laptop to use, some using tablets
- Javascript was the most obvious choice, as it runs everywhere
- But kids find it booooring….

A Bit of History

D as a teaching language

- Kids want to write games
- Roblox w/ Lua was the next adventure
- Nice base, but little opportunity for “regular” learning
- Spent more time dealing with the quirks of Roblox

A Bit of History

D as a teaching language

- Of course, I went to D after that.
- Kids without laptops borrowed them for class
- Started small, with text-based games (e.g. hangman)
- But the plan obviously was to move to graphical games

The Raylib Game Library

https://raylib.com

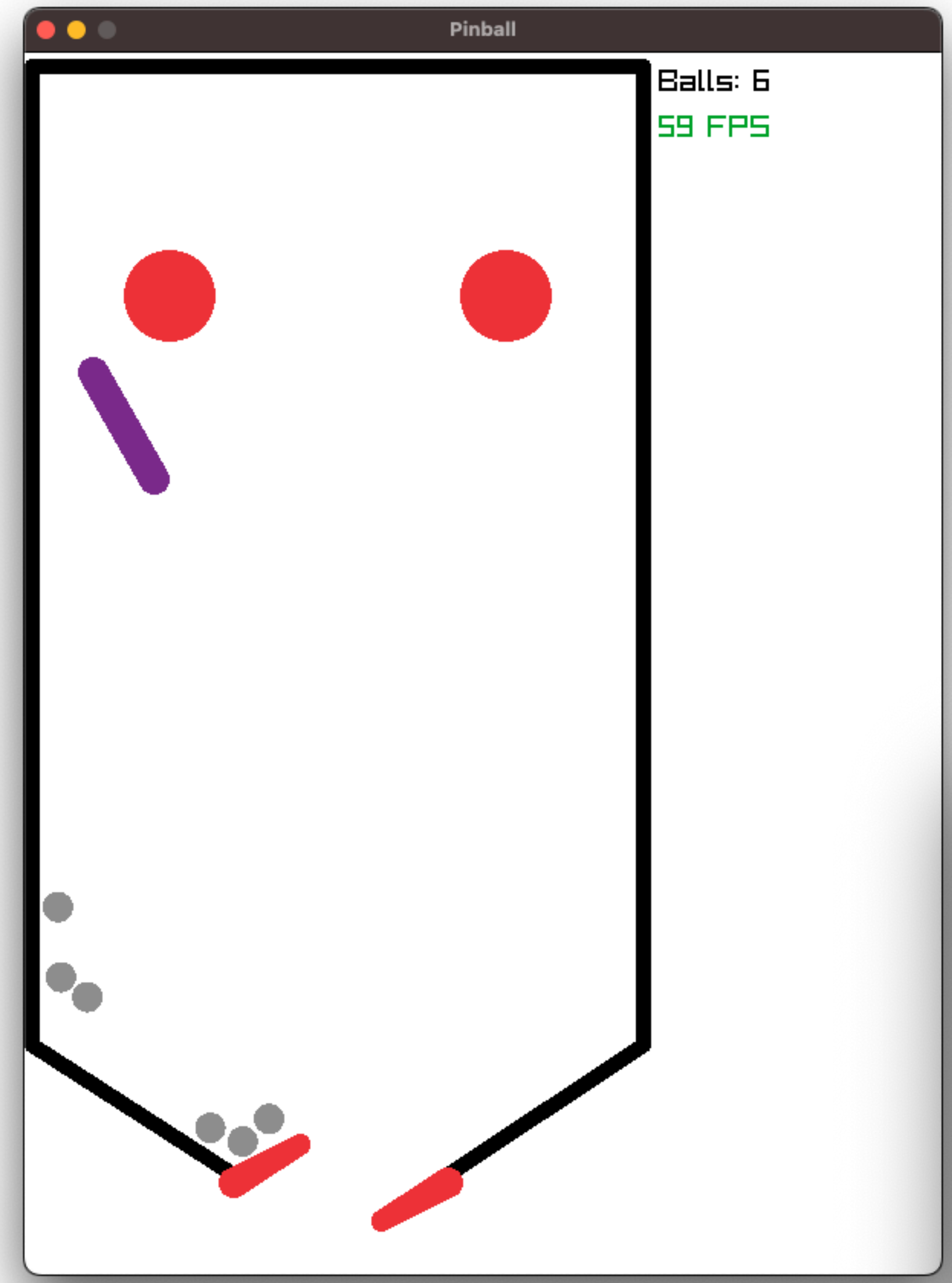
- Simple abstraction for writing games
- Written in C, but with existing D bindings
- A very nice introduction from Ki Rill on youtube: https://github.com/rillki/learn-dlang
- We could get up and running in 1 lesson, and make rudimentary drawings
- By using raylib, I can focus on the overall design of game development, and not get held back by complex systems.

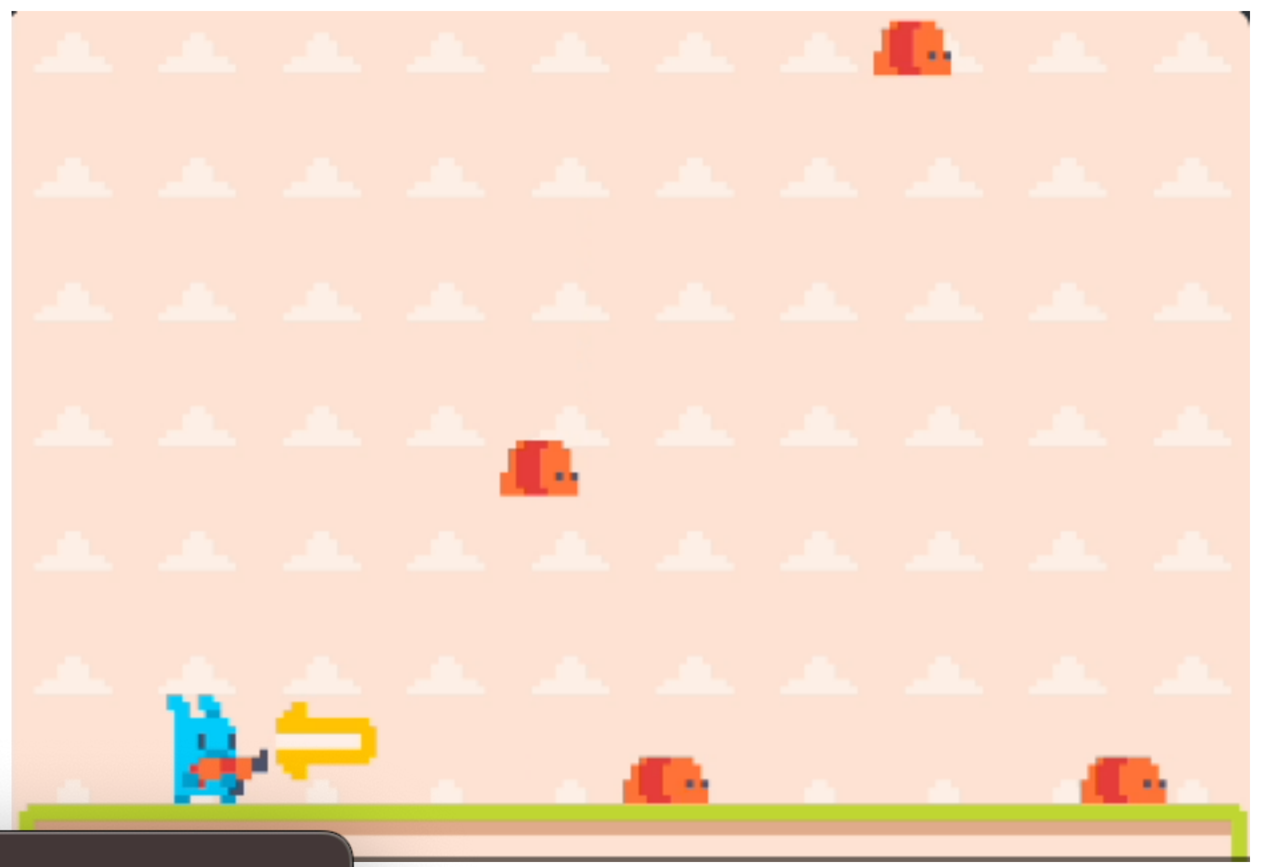
The Raylib-d binding

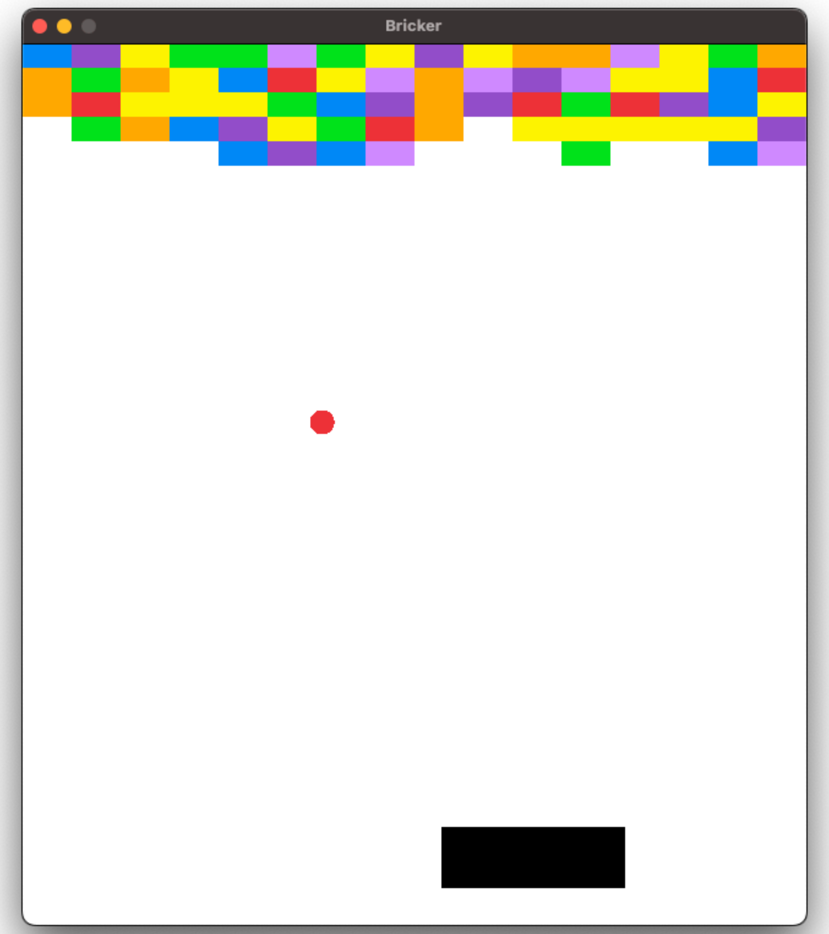
https://github.com/schveiguy/raylib-d

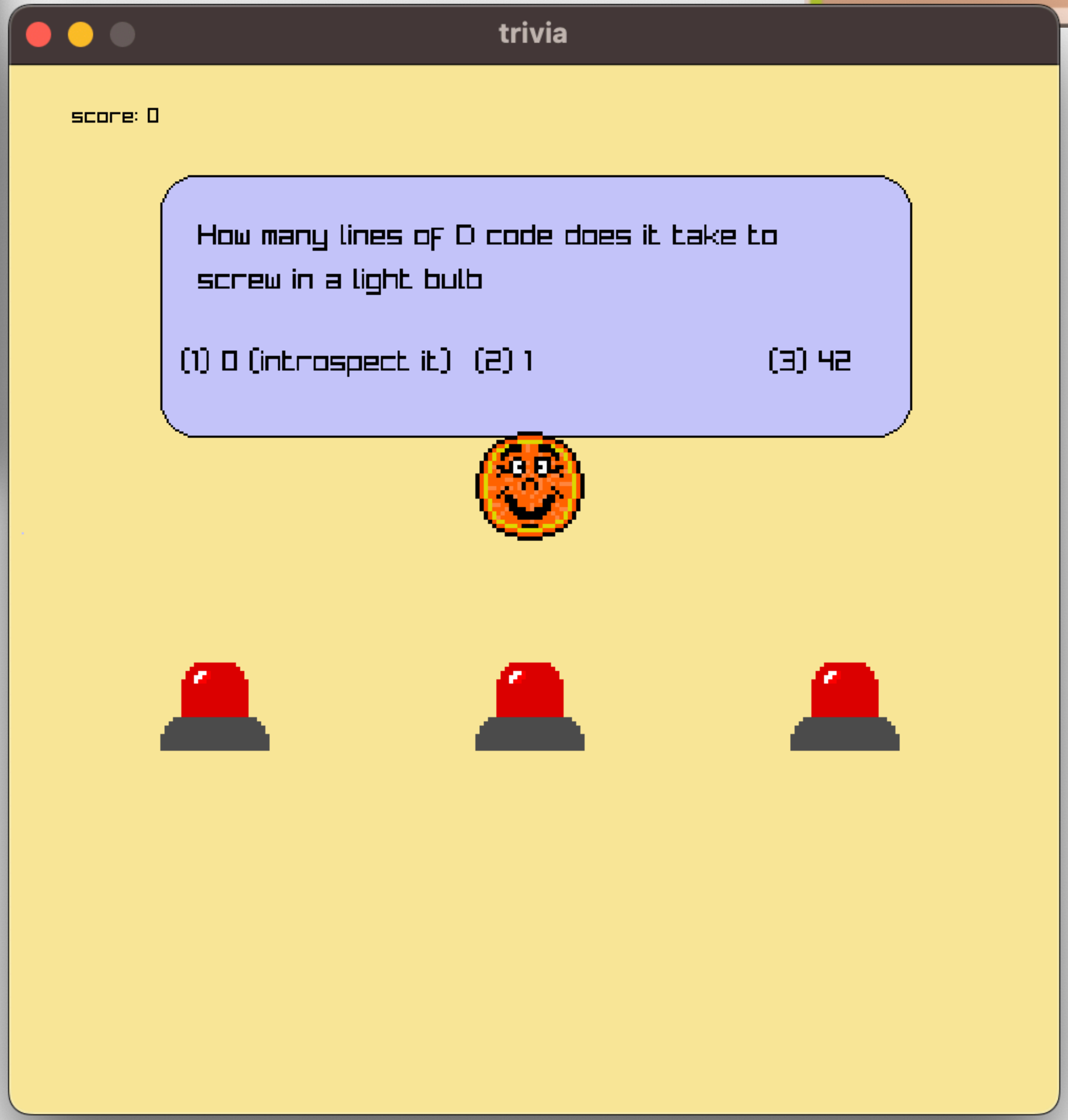
- raylib-d binding was provided by a GitHub user!
- They deleted their account :(
- But it’s open source, so I resurrected it >:)
- Now I am the maintainer (going on 4 revisions so far)
- With help from other users, I can build the binding in a matter of minutes

Raylib-d Examples!



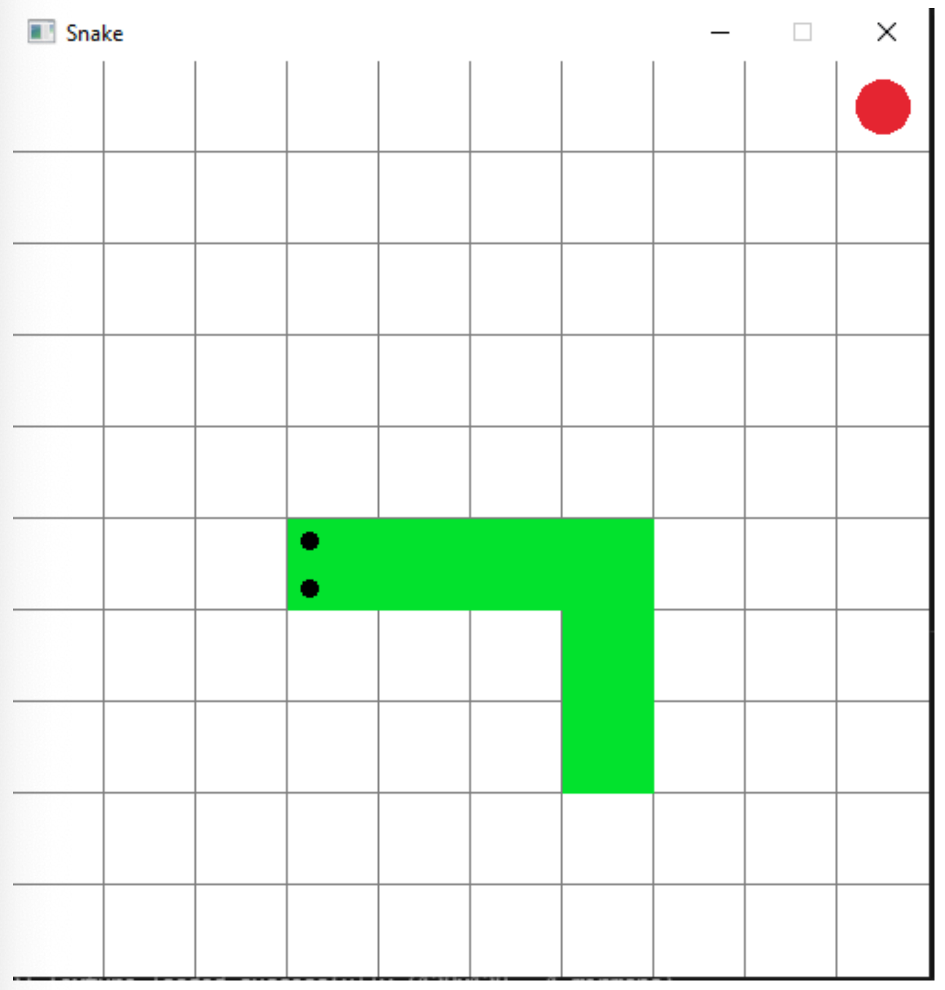






How many lines of D code does it take to screw in a light bulb

(1) 0 (introspect it) (2) 1 (3) 42



Stage 1 - Binding

Binding C From D

Easy

- D follows the C ABI for the platform of the system.
- No translation layers
- Requires C code is built using C compiler
- Dstep tool (https://github.com/jacob-carlborg/dstep) to generate bindings automatically

Binding C From D

dstep example

C:

```c
typedef struct {
    float x;
    float y;
    const char* name;
} label;

typedef struct {
    int width;
    int height;
    unsigned char buf[2048];
} pixelbuffer;

#define DEFAULT_WIDTH 500
#define DEFAULT_HEIGHT 500

void drawlabel(label lab, pixelbuffer buf);
```

D:

```d
extern (C):

struct label
{
    float x;
    float y;
    const(char)* name;
}

struct pixelbuffer
{
    int width;
    int height;
    ubyte[2048] buf;
}

enum DEFAULT_WIDTH = 500;
enum DEFAULT_HEIGHT = 500;

void drawlabel (label lab, pixelbuffer buf);
```

Binding C From D

Easy but messy

- C preprocessor macros are not so easy to translate.
  - Can basically just build whatever you want for the C compiler
  - Sometimes macros are part of the API
  - Non-hygienic
- Some ways to replace macros with mixins, or with inline/CTFE-able functions
- But not everything is doable…

Binding C From D

Creating the raylib-d binding

- Instructions for creating raylib-d are located in the generating.md file in the raylib-d repository
- Start with dstep
- Simple cleanup steps after the conversion
- Test against known projects!

Binding C From D

Creating the raylib-d binding

The most important part of creating bindings:

AUTOMATION!

Binding C From D

Automation is required!

- I don’t have time to do an in-depth analysis of the changes between versions
- C has no name mangling!
- Struct differences cannot be detected!
- Use Continuous Integration to prevent mistakes

Stage 2 - Improving The Binding

Doing the little things

Don’t use D to write C

- C libraries are written in C!
- But the D binding need not be.

```
Vector2 v1 = ...;
Vector2 v2 = ...;
// calculate velocity magnitude along the normal

// The C way
Vector2 v3 = Vector2Scale(
        Vector2Subtract(
            Vector2Add(Vector2Scale(mass, v1), Vector2Scale(mass, v2)),
            Vector2Scale(Vector2Subtract(v1, v2), mass * ballRestitution)
        ),
        1.0 / (mass + mass)
    );

// the D way (operator overloads)
auto v3 = (mass * v1 + mass * v2 - mass * ballRestitution * (v1 - v2)) / (mass + mass);
```

Doing the little things

Don’t use D to write C

- Enumerations in D are a type, in C they are always a constant
- **Key**board**Key.KEY**_COMMA
- Verbose, repetitive namespace — required for C, not for D
- Keyboard.Comma
- Automated enum generation easy to do with D.

Doing the little things

Don’t use D to write C

```d
enum KeyboardKey {
    KEY_COMMA,
    KEY_COLON
}

enum betterEnum(T, string newenum, string prefix) = (){
    string result = "enum " ~ newenum ~ " {\n";
    static foreach(m; __traits(allMembers, T))
    {
        static assert(m[0 .. prefix.length] == prefix);
        result ~= "    " ~ m[prefix.length .. $] ~ " = " ~
            T.stringof ~ "." ~ m ~ ",\n";
    }
    return result ~ "}\n";
}();

mixin(betterEnum!(KeyboardKey, "Keyboard", "KEY_"));

void main()
{
    auto k = Keyboard.COMMA;
}
```

Doing the little things

Don’t use D to write C

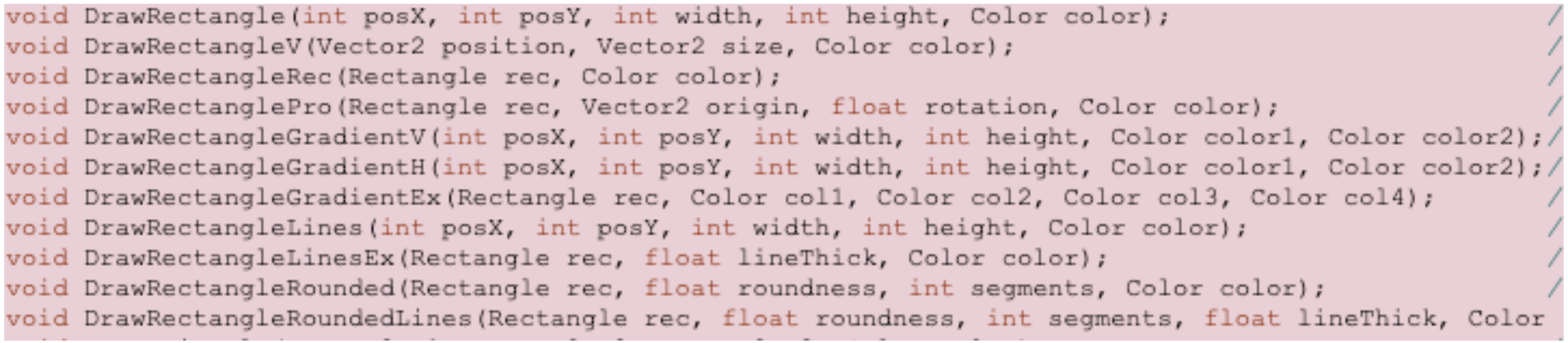
- Wrapping C abstractions
- Adding UFCS and methods
- Overloading works!
- Strings….

Stage 3 - Porting

Using C from D is still not great

C has so much cruft…

- C strings are horrendous to use in D
- D has fantastic string manipulation, but using them with raylib is pain.
- Overloads with C are awkward…



```c
void DrawRectangle(int posX, int posY, int width, int height, Color color);                      /
void DrawRectangleV(Vector2 position, Vector2 size, Color color);                                /
void DrawRectangleRec(Rectangle rec, Color color);                                               /
void DrawRectanglePro(Rectangle rec, Vector2 origin, float rotation, Color color);               /
void DrawRectangleGradientV(int posX, int posY, int width, int height, Color color1, Color color2);/
void DrawRectangleGradientH(int posX, int posY, int width, int height, Color color1, Color color2);/
void DrawRectangleGradientEx(Rectangle rec, Color col1, Color col2, Color col3, Color col4);     /
void DrawRectangleLines(int posX, int posY, int width, int height, Color color);                 /
void DrawRectangleLinesEx(Rectangle rec, float lineThick, Color color);                          /
void DrawRectangleRounded(Rectangle rec, float roundness, int segments, Color color);            /
void DrawRectangleRoundedLines(Rectangle rec, float roundness, int segments, float lineThick, Color
```

Using C from D is still not great

The string problem

- Raylib has TextFormat
- D has std.format.format, and std.conv.text
- With new programmers, D string interpolation functions are easy to get right, and are understandable.
- But still must be converted. Into a pointer…
- Wrapping is a possibility, but has some drawbacks.

Needing to use C

C as a dependency is problematic

- C cannot (yet) be compiled by dub
- Including pre-built binaries does not scale
- raylib-d binding contains a tool now to install these binaries.
- Even when using all the tools, I can’t anticipate all issues.
- Holy grail for users of raylib in D: dub add raylib, and build your application.

Introducing draylib!

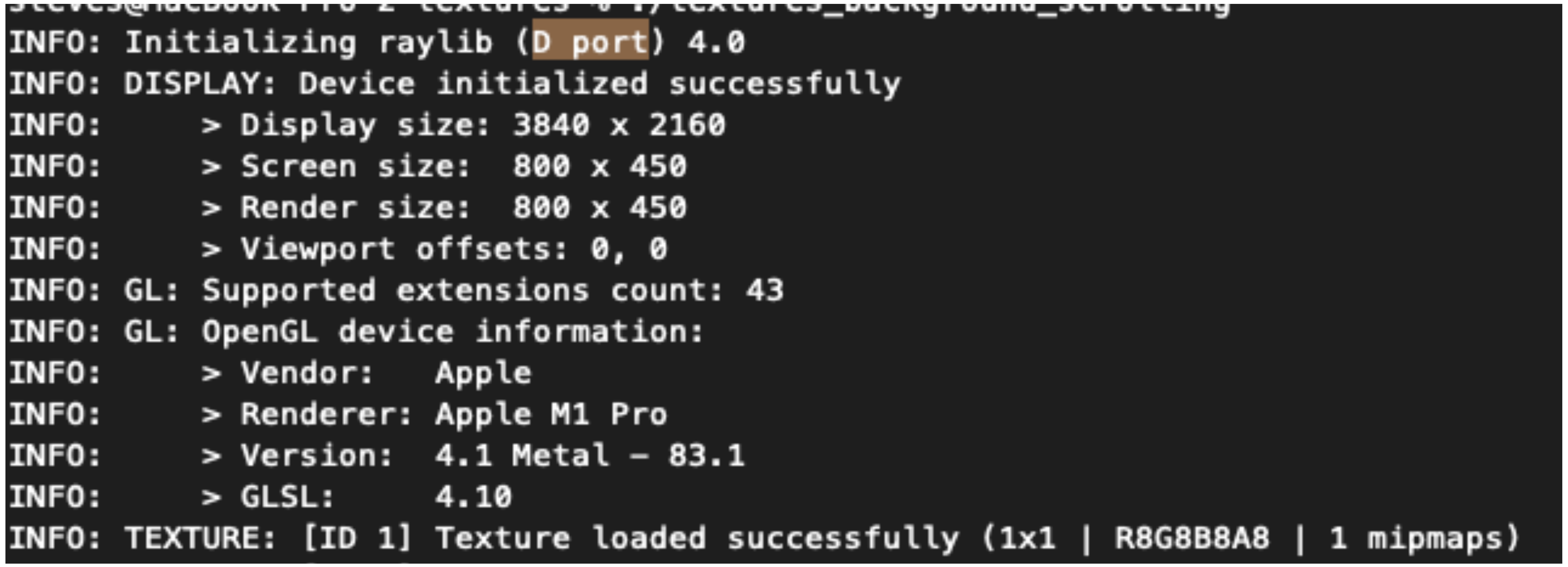
https://github.com/schveiguy/draylib

- Goal: a complete port of raylib C code to D
- No more need for C compiler or custom prebuilt libraries.
- 2 developers, @realDoigt and @schveiguy
- Once fully ported the API will be “D-ified”
- Keep the simplicity of C raylib, with the experience of using D.
- Raylib is clean straightforward C code. How hard could it be?

Copy C code - build as D

Not so easy…

- First module - rcore.c
- Comment out code in rcore.c, and copy the code into rcore.d.
- Compile, fix errors, add more code, compile, fix errors, etc.
- After a set of functions is ported, build against the examples, see that they still work.



```
INFO: Initializing raylib (D port) 4.0
INFO: DISPLAY: Device initialized successfully
INFO:     > Display size: 3840 x 2160
INFO:     > Screen size:  800 x 450
INFO:     > Render size:  800 x 450
INFO:     > Viewport offsets: 0, 0
INFO: GL: Supported extensions count: 43
INFO: GL: OpenGL device information:
INFO:     > Vendor:   Apple
INFO:     > Renderer: Apple M1 Pro
INFO:     > Version:  4.1 Metal - 83.1
INFO:     > GLSL:     4.10
INFO: TEXTURE: [ID 1] Texture loaded successfully (1x1 | R8G8B8A8 | 1 mipmaps)
```

Copy C code - build as D

Not so easy…

- Annoying little things:
  - `NULL` to `null`
  - `ptr->mem` to `ptr.mem`
  - `int arr[5]` to `int[5] arr`
  - `unsigned int` to `uint`
  - `unsigned char` to `ubyte`
  - etc…
  - "Translating C to D": https://dconf.org/2022/online/#dennisk

Copy C code - build as D

dependencies needed

- Still building C code, so just use dstep
- Some small modules can just port quickly
- A large complex dependency is `glfw`.
- Just use bindbc binding, and keep building with C.
  https://code.dlang.org/packages/bindbc-glfw
- Wait, there's a glfw-d project! How did that happen? hm….
  https://code.dlang.org/packages/glfw-d

rcore.d fully ported!

For now, only desktop supported

- Time to port (several functions at a time): June to October.
- ~7500 lines of code
- Sporadic work, estimate about 40-80 hours. About 125 LOC/hour

How much is left?

Wait, there’s more…

- raylib still has more modules to do:
  - rtextures.c: 4800 LOC
  - rtext.c: 2100 LOC
  - rmodels.c: 5900 LOC
  - And more…
- raylib external libraries
  - stb_image.h: 8000 LOC
  - stb_vorbis.h: 5500 LOC
  - miniaudio.h: 70000 LOC(!)
  - And more…
- Manual porting, not going to cut it.

Side quest - Leave some C

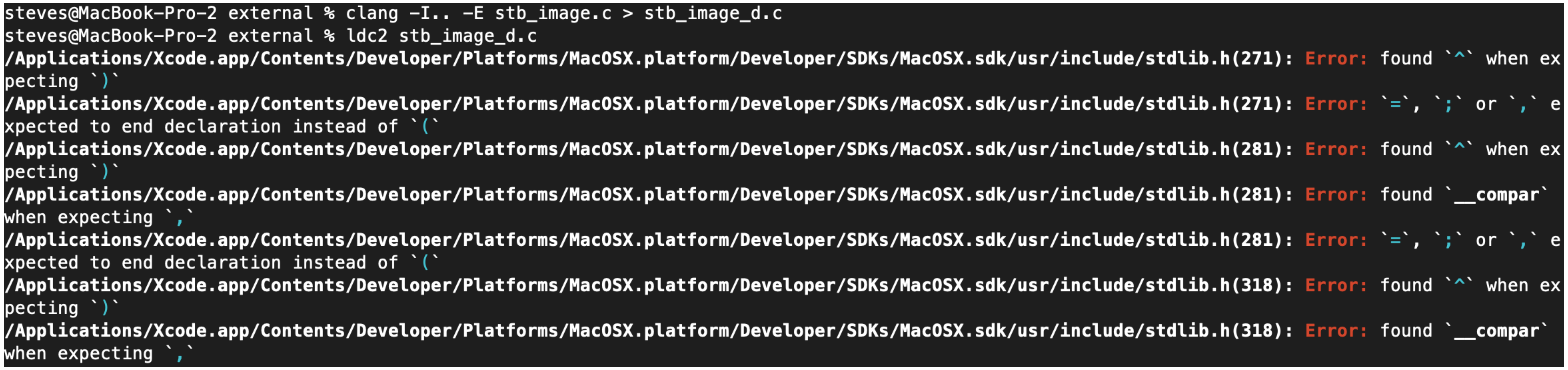
External dependencies don’t need to be D

- All these external modules/libraries don’t need to be ported to D.
- There are no public API interfaces to these, they are *implementation details*.
- Maybe leave some of these as being compiled C code that just have bindings?

Side quest - Leave some C

ImportC?

- The latest DMD/LDC has a built-in C compiler!
- Just compile the implementation detail dependencies with ImportC, and no external C compiler required!
- But… it doesn't work here.



```
steves@MacBook-Pro-2 external % clang -I.. -E stb_image.c > stb_image_d.c
steves@MacBook-Pro-2 external % ldc2 stb_image_d.c
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdlib.h(271): Error: found `^` when ex
pecting `)`
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdlib.h(271): Error: `=`, `;` or `,` e
xpected to end declaration instead of `(`
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdlib.h(281): Error: found `^` when ex
pecting `)`
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdlib.h(281): Error: found `__compar`
when expecting `,`
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdlib.h(281): Error: `=`, `;` or `,` e
xpected to end declaration instead of `(`
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdlib.h(318): Error: found `^` when ex
pecting `)`
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdlib.h(318): Error: found `__compar`
when expecting `,`
```

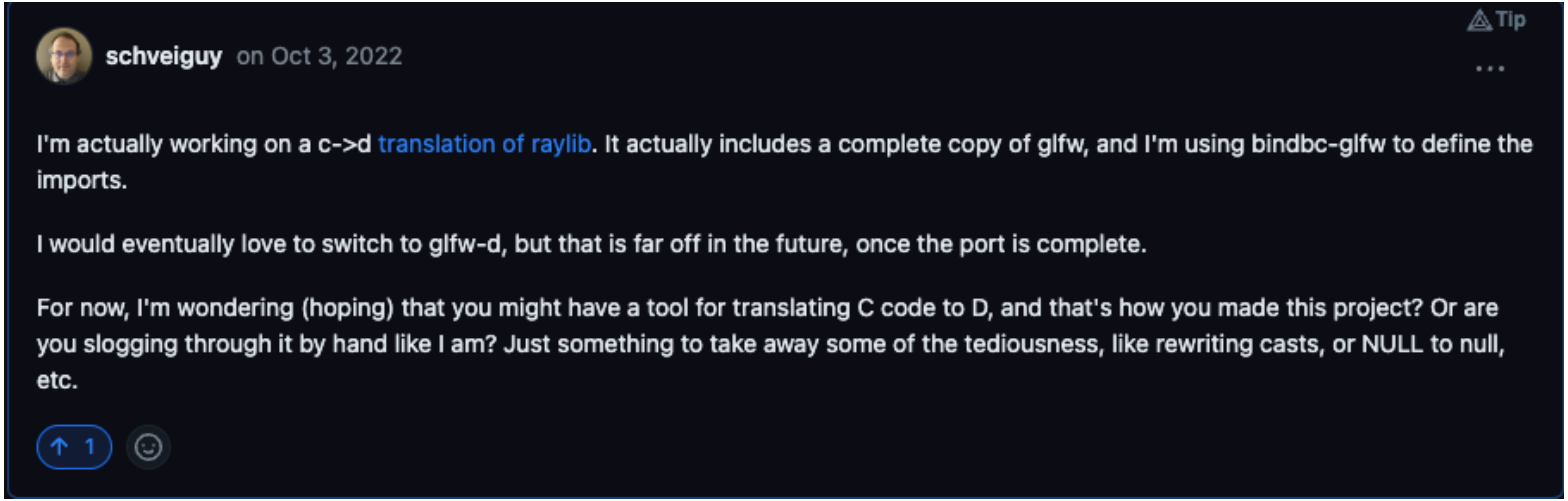
Still need C…

Resigned to the reality

- Use pre-built C object files?
- Invoke the C compiler from dub?
- dub does a poor job managing external artifacts.
- There must be something better. How *did* that glfw-d port happen?

A miracle happens

How did Dennis do it?


Tip

schveiguy on Oct 3, 2022

I'm actually working on a c->d translation of raylib. It actually includes a complete copy of glfw, and I'm using bindbc-glfw to define the imports.

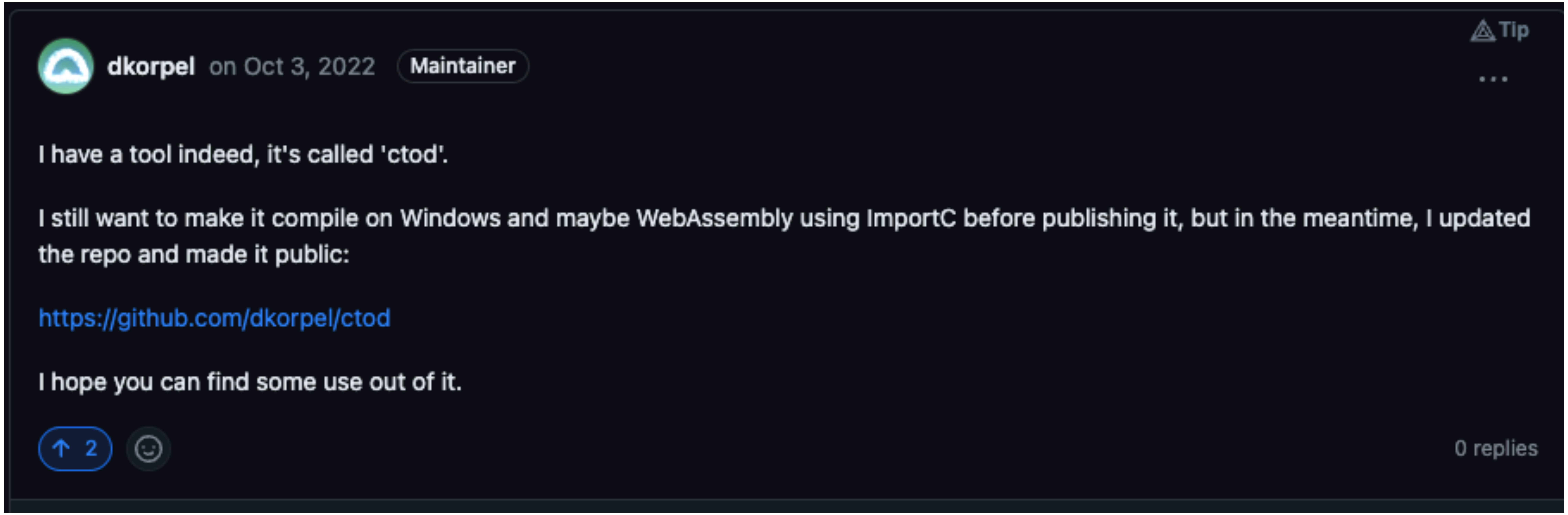
I would eventually love to switch to glfw-d, but that is far off in the future, once the port is complete.

For now, I'm wondering (hoping) that you might have a tool for translating C code to D, and that's how you made this project? Or are you slogging through it by hand like I am? Just something to take away some of the tediousness, like rewriting casts, or NULL to null, etc.

1

A miracle happens

Answer: the smart way



I have a tool indeed, it's called 'ctod'.

I still want to make it compile on Windows and maybe WebAssembly using ImportC before publishing it, but in the meantime, I updated the repo and made it public:

https://github.com/dkorpel/ctod

I hope you can find some use out of it.

Stage 4 - Making Porting Tools

ctod

https://github.com/dkorpel/ctod

- Based on TreeSitter, to allow for “not quite parsable” code
- Handles lots of minutia!
- Maybe a solution for porting these “implementation details”?

Attempt at stb_image.h

Needed for rtextures.c

- ctod gets much of the way there!
- During this process, filed almost 30 issues/bugs against ctod
- Bugs get fixed!
- But now, I have to run the process again…

Porting C From D

The most important part of porting C code to D:

AUTOMATION!

ctod is not enough

Macros…

```
#ifdef _WIN32
    #ifdef IS_DYNAMIC
        #define linkage __declspec(dllexport)
    #endif
#endif

linkage void foo()
{
}
```

```
module linkage;
@nogc nothrow:
extern(C): __gshared:
version (Windows) {
    version (IS_DYNAMIC) {
        enum linkage = __declspec(dllexport);
    }
}

linkage void foo()
{
}
```

“Solving” the macro problem

Just expand the macros

- For the “implementation details” files, we don’t care about versioning, or code aesthetics.
- Once it is ported, it is done.
- Focus first on getting it working on one platform
- Just run the part of the preprocessor that replaces macros.

“Solving” the macro problem

Just expand the macros

- gcc preprocessor has the -d switch — for debugging
- -dDI keeps the existing macro definitions and includes (so ctod can see them), and also keeps the include headers, but *also expands macros*.
- The -C switch keeps comments
- It also outputs lots of directives whenever it switches files.
- Using the output, we can get the macro expansion we need, and get back to the “original” code by removing the included files.

But what about other compilers?

clang and MSVC?

- clang does not have the -d option :(
- Neither does MSVC :(
- But both have an option to keep comments!
- I can work with this >:)

Introducing cpptool!

https://github.com/schveiguy/cpptool

- Step 1: detect all # directives. Add a pair of cpptool-special comments of the form:

```
//>> 1 #define foo bar
#define foo bar
//<< 1 #define foo bar
```

- The `//<<` and `//>>` are specialized comments that show the code that will be removed by the preprocessor.
- The 1 is an id to make sure we don't see it twice. We need both because it might be on either side of a conditional clause

Introducing cpptool!

https://github.com/schveiguy/cpptool

```
#include "foo.h"

#define BEGIN(fn) int fn() {
#define END }

BEGIN(main)
    int x = bar;
END
```

```
//CPPTOOL cpptool_cpptool_tmp.c
//>> 0 #include "foo.h"
#include "foo.h"
//<< 0 #include "foo.h"

//>> 1 #define BEGIN(fn) int fn() {
#define BEGIN(fn) int fn() {
//<< 1 #define BEGIN(fn) int fn() {
//>> 2 #define END }
#define END }
//<< 2 #define END }

BEGIN(main)
    int x = bar;
END
```

Introducing cpptool!

https://github.com/schveiguy/cpptool

- Step 2: run the system preprocessor on the modified file.

```
//CPPTOOL cpptool_cpptool_tmp.c
//>> 0 #include "foo.h"
#include "foo.h"
//<< 0 #include "foo.h"

//>> 1 #define bar 5
#define bar 5
//<< 1 #define bar 5

int main() {
    int x = bar;
}
```

```
# 1 "cpptool_cpptool_tmp.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 414 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "cpptool_cpptool_tmp.c" 2
//CPPTOOL cpptool_cpptool_tmp.c
//>> 0 #include "foo.h"
# 1 "./foo.h" 1
// this is foo.h!
# 4 "cpptool_cpptool_tmp.c" 2
//<< 0 #include "foo.h"

//>> 1 #define BEGIN(fn) int fn() {

//<< 1 #define BEGIN(fn) int fn() {
//>> 2 #define END }

//<< 2 #define END }

int main() {
    int x = 6;
}
```

Introducing cpptool!

https://github.com/schveiguy/cpptool

- Step 3: run cpptool on the result in “recover” mode to get back to where we were originally, but with macros expanded.

```
# 1 "cpptool_cpptool_tmp.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 414 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "cpptool_cpptool_tmp.c" 2
//CPPTOOL cpptool_cpptool_tmp.c
//>> 0 #include "foo.h"
# 1 "./foo.h" 1
// this is foo.h!
# 4 "cpptool_cpptool_tmp.c" 2
//<< 0 #include "foo.h"

//>> 1 #define BEGIN(fn) int fn() {

//<< 1 #define BEGIN(fn) int fn() {
//>> 2 #define END }

//<< 2 #define END }

int main() {
    int x = 6;
}
```

```
#include "foo.h"

#define BEGIN(fn) int fn() {

#define END }


int main() {
    int x = 6;
}
```

Introducing cpptool!

https://github.com/schveiguy/cpptool

- Step 4: run ctod on the result!

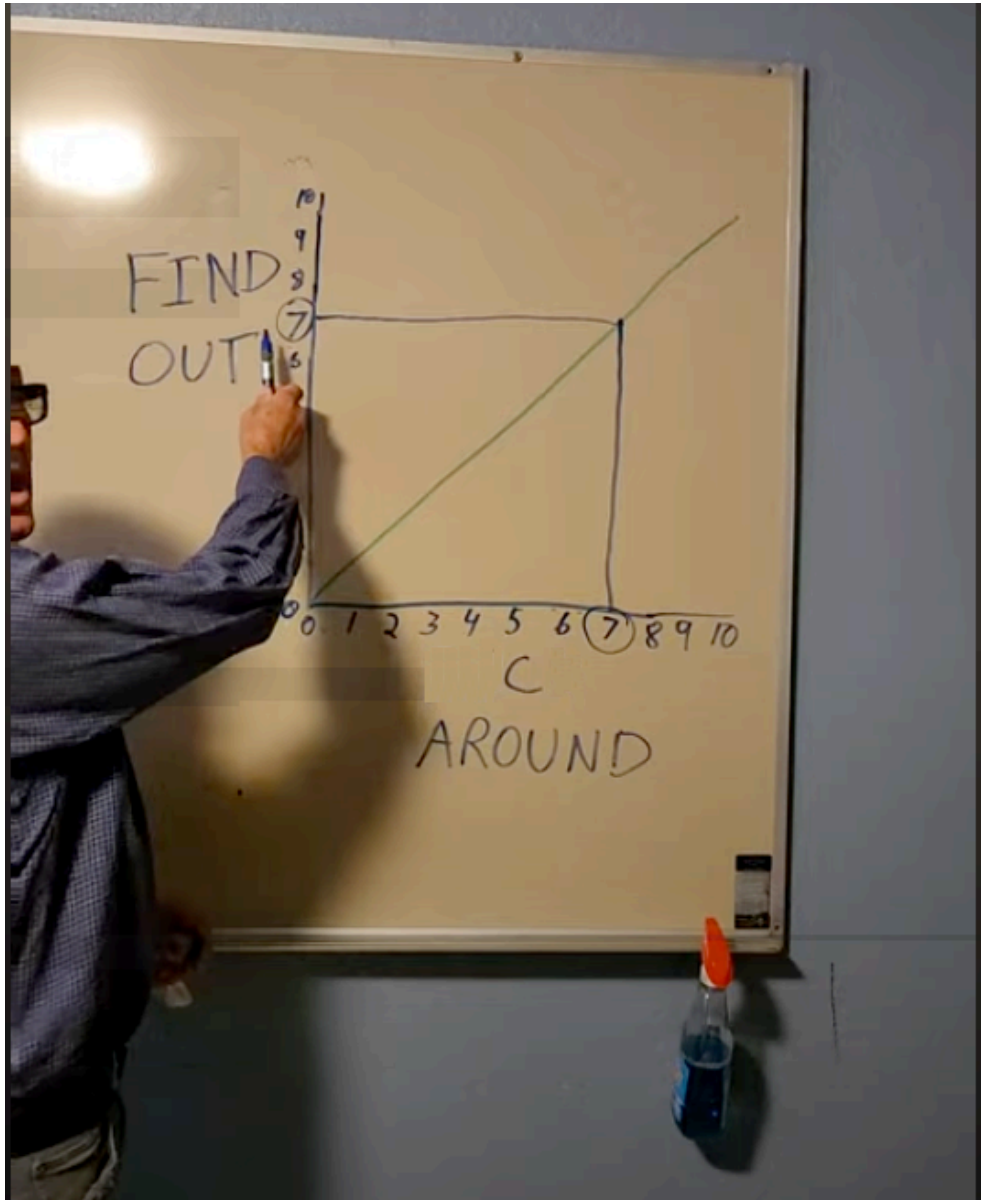
```
#include "foo.h"

#define BEGIN(fn) int fn() {

#define END }


int main() {
    int x = 6;
}
```

```
module cpptool;
@nogc nothrow:
extern(C): __gshared:
public import foo;

enum string BEGIN(string fn) = `int fn() {}`;

enum END = };


int main() {
    int x = 6;
}
```

Introducing cpptool!

https://github.com/schveiguy/cpptool

- Step 5: find out all the problems.
- #defines for things we expect
- #defines for things we don't expect
- Add a way to skip #defines I don't want to replace



Introducing cpptool!

https://github.com/schveiguy/cpptool

- Step 6: fix the process, do it again.
- AUTOMATION IS IMPORTANT!

Things that can’t be automated

O C how annoying art thou?

- Implicit integer conversion
- `0` as `NULL`
- initialization with `{ 0 }`
- comma expressions to cram multiple statements into one.

Things that can’t be automated

Yeah, D, you too.

- sizeof is used a lot for things like lengths, but stored to int

```
int delays_size = 0;

…

delays_size = layers * sizeof(int);
```

```
int delays_size = 0;

…

delays_size = layers * int(int.sizeof);
```

Things that can’t be automated

Yeah, D, you too.

- Handling a comma expression

```
stbiw__zlib_huff(j+257);
```

Things that can’t be automated

Yeah, D, you too.

- Handling a comma expression

```
((j+257) <= 143 ? (bitbuf |=
(stbiw__zlib_bitrev(0x30 + (j+257),8)) <<
bitcount, bitcount += (8), (out_ =
stbiw__zlib_flushf(out_, &bitbuf, &bitcount))) :
(j+257) <= 255 ? (bitbuf |=
(stbiw__zlib_bitrev(0x190 + (j+257)-144,9)) <<
bitcount, bitcount += (9), (out_ =
stbiw__zlib_flushf(out_, &bitbuf, &bitcount))) :
(j+257) <= 279 ? (bitbuf |= (stbiw__zlib_bitrev(0
+ (j+257)-256,7)) << bitcount, bitcount += (7),
(out_ = stbiw__zlib_flushf(out_, &bitbuf,
&bitcount))) : (bitbuf |= (stbiw__zlib_bitrev(0xc0
+ (j+257)-280,8)) << bitcount, bitcount += (8),
(out_ = stbiw__zlib_flushf(out_, &bitbuf,
&bitcount))));
```

```
(j+257) <= 143 ? () {
    bitbuf |= (stbiw__zlib_bitrev(0x30 + (j+257),8)) <<
bitcount;
    bitcount += (8);
    return out_ = stbiw__zlib_flushf(out_, &bitbuf, &bitcount);
}()
: (j+257) <= 255 ? (){
    bitbuf |= (stbiw__zlib_bitrev(0x190 + (j+257)-144,9)) <<
bitcount;
    bitcount += (9);
    return out_ = stbiw__zlib_flushf(out_, &bitbuf, &bitcount);
}()
: (j+257) <= 279 ? () {
    bitbuf |= (stbiw__zlib_bitrev(0 + (j+257)-256,7)) <<
bitcount;
    bitcount += (7);
    return out_ = stbiw__zlib_flushf(out_, &bitbuf, &bitcount);
}()
: () {
    bitbuf |= (stbiw__zlib_bitrev(0xc0 + (j+257)-280,8)) <<
bitcount;
    bitcount += (8);
    return out_ = stbiw__zlib_flushf(out_, &bitbuf, &bitcount);
}();
```

Things that can’t be automated

Yeah, D, you too.

- No equivalent to `__thread`

```c
static
#ifdef STBI_THREAD_LOCAL
__thread
#endif
const char *stbi__g_failure_reason;
```

```d
__gshared:
struct stbi__g_failure_reason_holder
{
    static const(char)* v;
}
alias stbi__g_failure_reason =
stbi__g_failure_reason_holder.v;
```

Things that can’t be automated

Yeah, D, you too.

- Variable shadowing
- Easy to fix — mostly these are indexes/single letter things. Just add a `1` to the variable name.

Things that can’t be automated

Yeah, D, you too.

- Unreachable statements 😡

```
switch(...) {
    ...
    default: STBI_ASSERT(0);
             STBI_FREE(data);
             STBI_FREE(good);
             return stbi__errpuc(“unsupported",
                    "Unsupported format conversion");
}
```

```
switch(...) {
    ...
    default: assert(0);
             free(data);
             free(good);
             return (cast(ubyte*)cast(size_t)
                (stbi__err("unsupported")?null:null));
}
```

Things that can’t be automated

Yeah, D, you too.

- Unreachable statements 😡

```
switch(...) {
    ...
    default: STBI_ASSERT(0);
             STBI_FREE(data);
             STBI_FREE(good);
             return stbi__errpuc(“unsupported",
                    "Unsupported format conversion");
}
```

```
switch(...) {
    ...
    default: assert(0);
             //free(data);
             //free(good);
             //return (cast(ubyte*)cast(size_t)
             //   (stbi__err("unsupported")?null:null));
}
```

Let’s talk about #ifdefs

No good options

- If the #define comes from the makefile, it’s more like a version
- If the #define is set in a file, it’s more like an enum
- C uses a mechanism of #defining an identifier (like a version), and then #including a header to affect it (like an enum)
- dub projects don’t have a good way to push config files to dependencies.
- C allows #defines from the command line that are not just “define this version”.

  ```
  #ifndef RL_MALLOC
      #define RL_MALLOC malloc
  #endif
  ```
- C has #undef!

Some ugliness with cpptool

it’s implementation details, who cares?

- All spacing is compressed into one space
- line continuations are concatenated
- Lots and lots of extra empty lines.
- Inactive portions are blank.
- Overabundance of parentheses

Dealing with multiple platforms

Need some more automation!

- Using the ids from the instrumented file, match up sections?
- Use diff tools to see differences in macro replacements?
- Some code explicitly uses compiler intrinsics (like SIMD instructions), harder to port?
- Could really use a working ImportC!

rtextures.d

It finally happened! (on MacOS)

- The following files were ported using cpptool and ctod:
  - stb_image.h
  - stb_image_write.h
  - stb_image_resize.h
- rtextures.c was ported just using ctod.
- Total time (minus all the tool dev time) probably around 2-3 hours.

Stage 5 - Improve the API

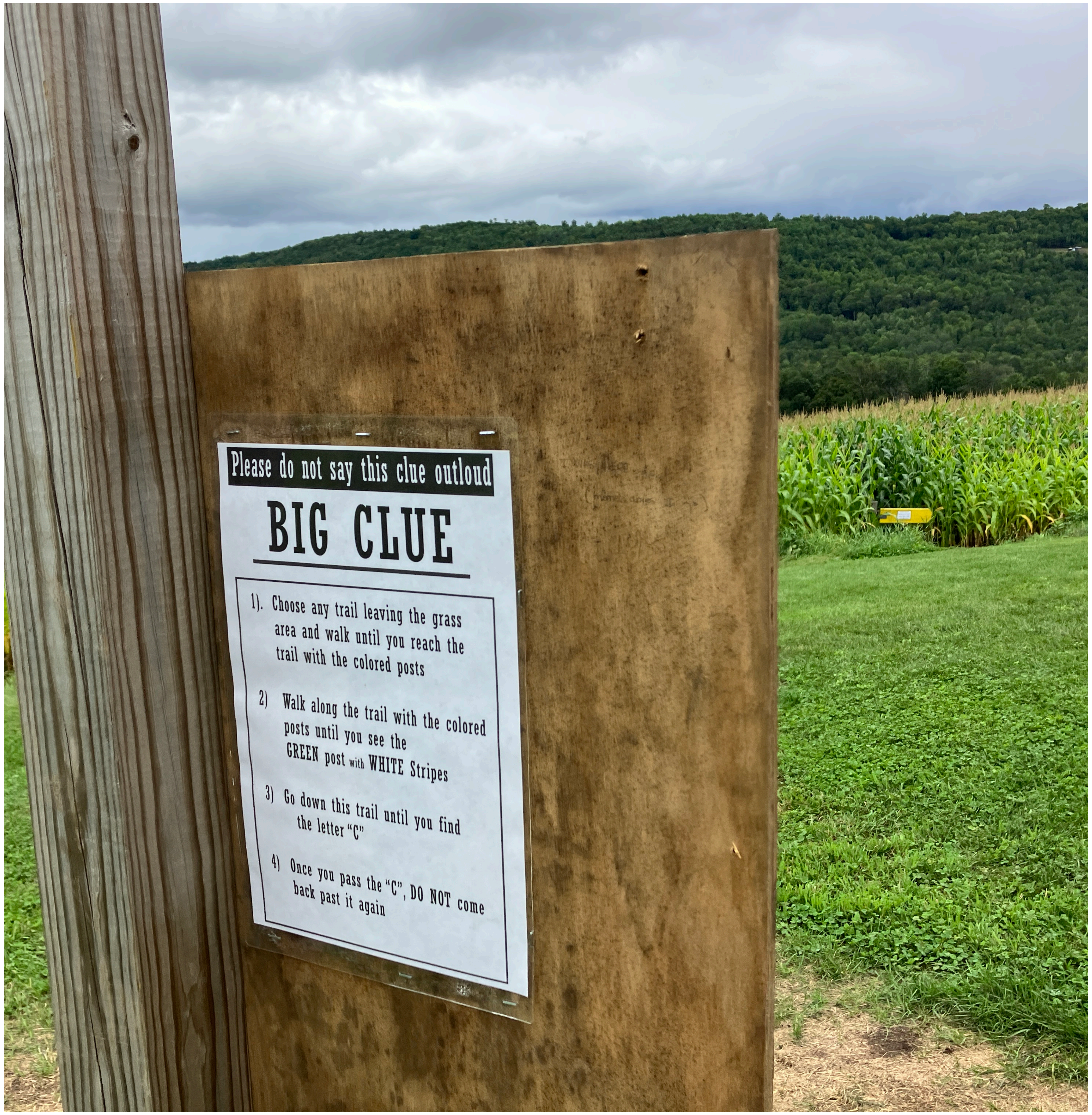
For the future!

Once it’s ported…

- Keep the C API (why not?)
- First to go: C strings
- Remove betterC as a requirement
- Memory safety?
- `Rectangle.draw` instead of `DrawRectangle`?
- Utilize constructors to aid in making types
- Examine changes that have happened since raylib 4.0.0, maybe include some.
- Porting guide for people who use raylib-d

Thank you!

NOTE: Portions of this presentation are known to the State of **C**alifornia to cause cancer, birth defects, or other reproductive harm.



Please do not say this clue outloud

BIG CLUE

1). Choose any trail leaving the grass area and walk until you reach the trail with the colored posts

2) Walk along the trail with the colored posts until you see the GREEN post with WHITE Stripes

3) Go down this trail until you find the letter "C"

4) Once you pass the "C", DO NOT come back past it again

Stripes

3) Go down this trail until you find the letter “C”

4) Once you pass the “C”, DO NOT come back past it again