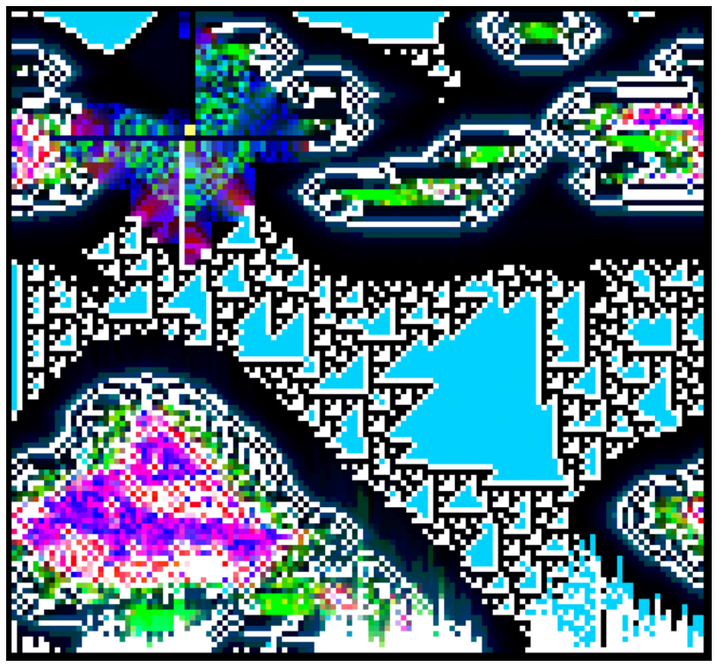
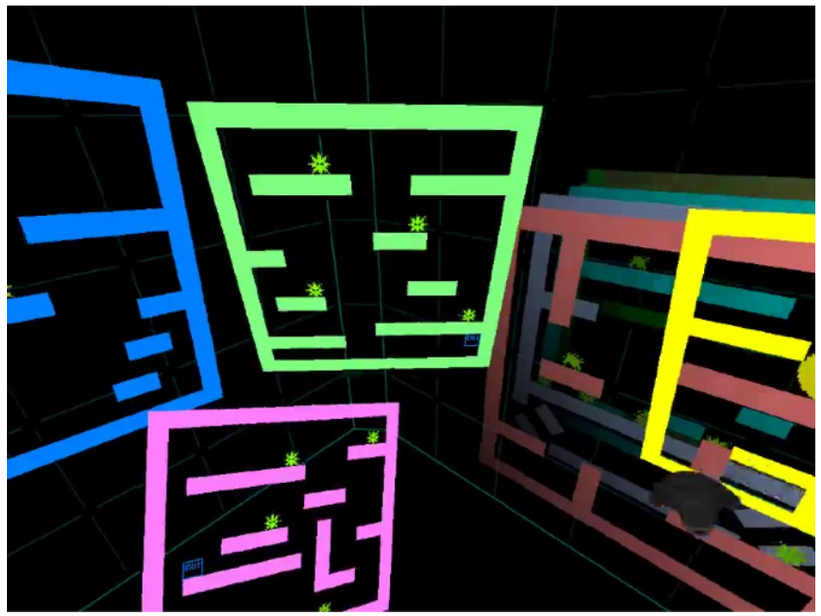
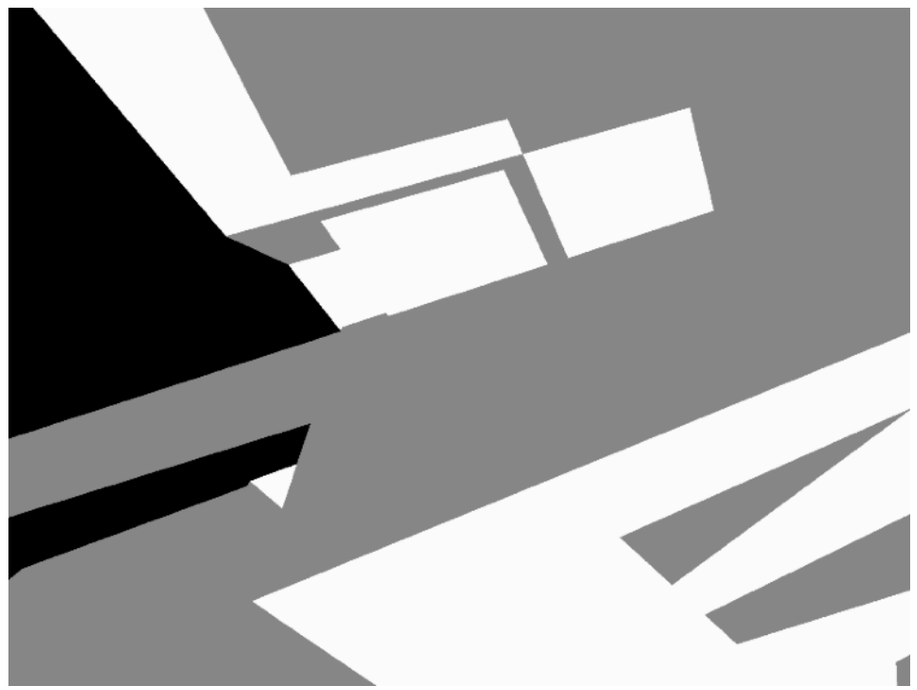
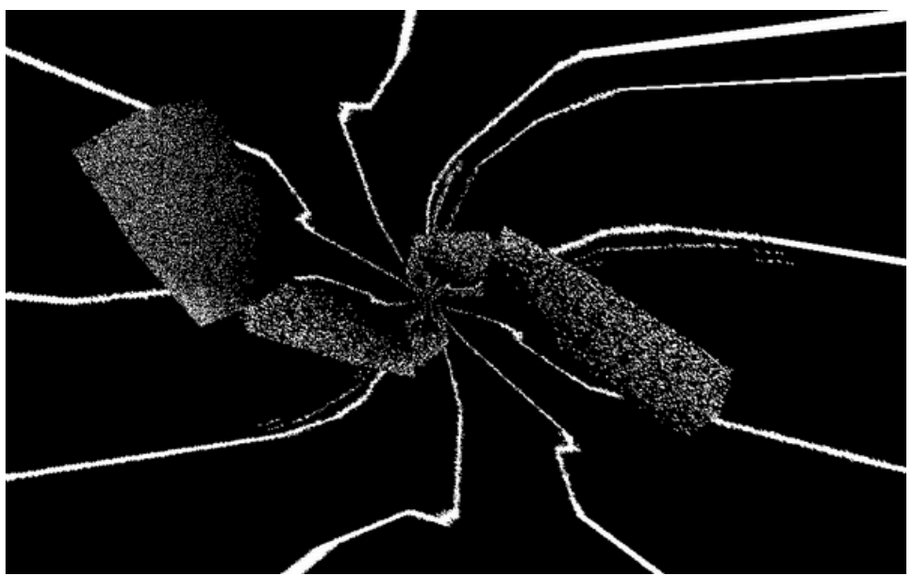
Making a programming language is surprisingly easy and fun

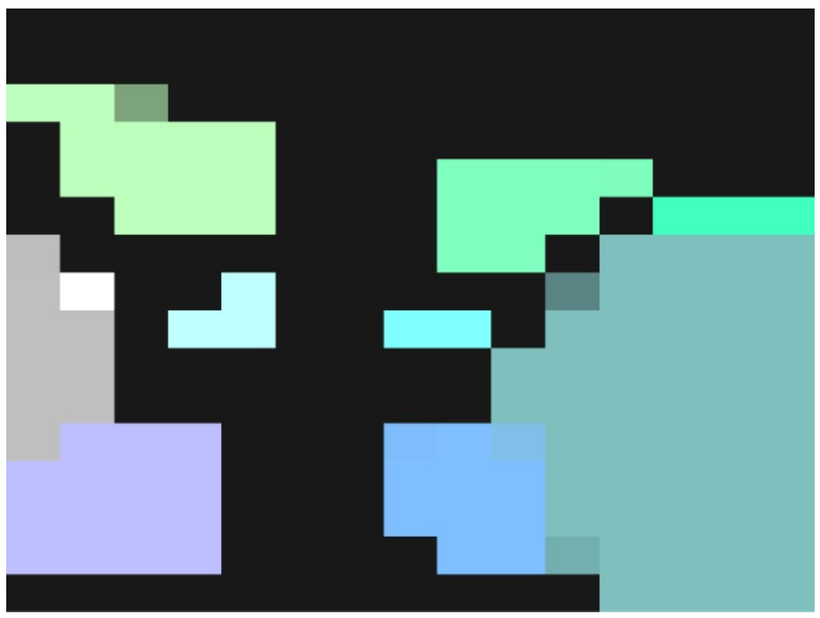
My name is Andi and I break things

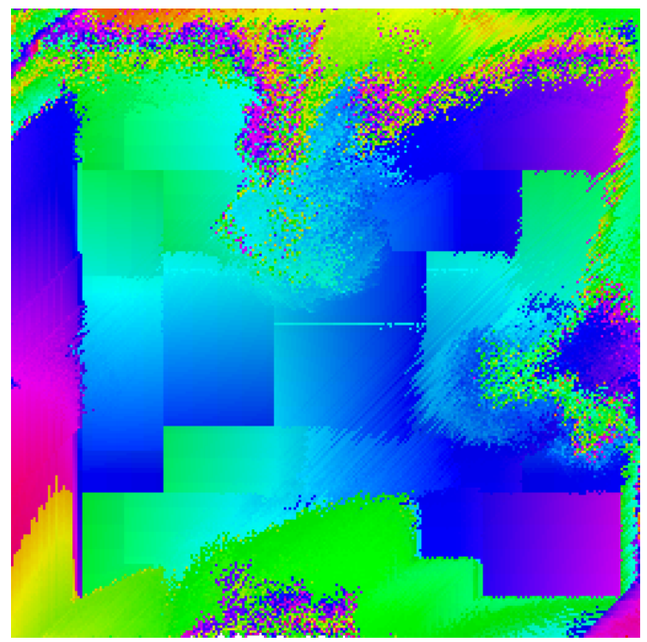












Emily

```
width = 80

foreach ^upto ^perform = {
    counter = 0
    while ^(counter < upto) ^( perform counter; counter = counter + 1; )
}

inherit ^class = [ parent = class ]

line = [                                # Object for one line of printout
    createFrom ^old = {
        foreach width ^at { # Rule 135
            final = width - 1
            here   = old at
            before = old ( at == 0 ? final : at - 1 )
            after  = old ( at == final ? 0 : at + 1 )
            this.append: ( here && before && after ) \
                    || !( here || before || after )
        }
    }
    print ^ = {
        this.each ^cell { print: cell ? "*" : " " }
        println ""                                          # Next line
    }
]

repeatWith ^old = {  # Repeatedly print a line, then generate a new one
    do: old.print
    new = inherit line
    new.createFrom old
    repeatWith new
}

starting = inherit line        # Create a starting line full of garbage
next = 1
foreach width ^at (
    starting.append: at != next
    if (at == next) ^( next = next * 2 )
)
repeatWith starting
```

Why make a programming language?

- Get superpowers

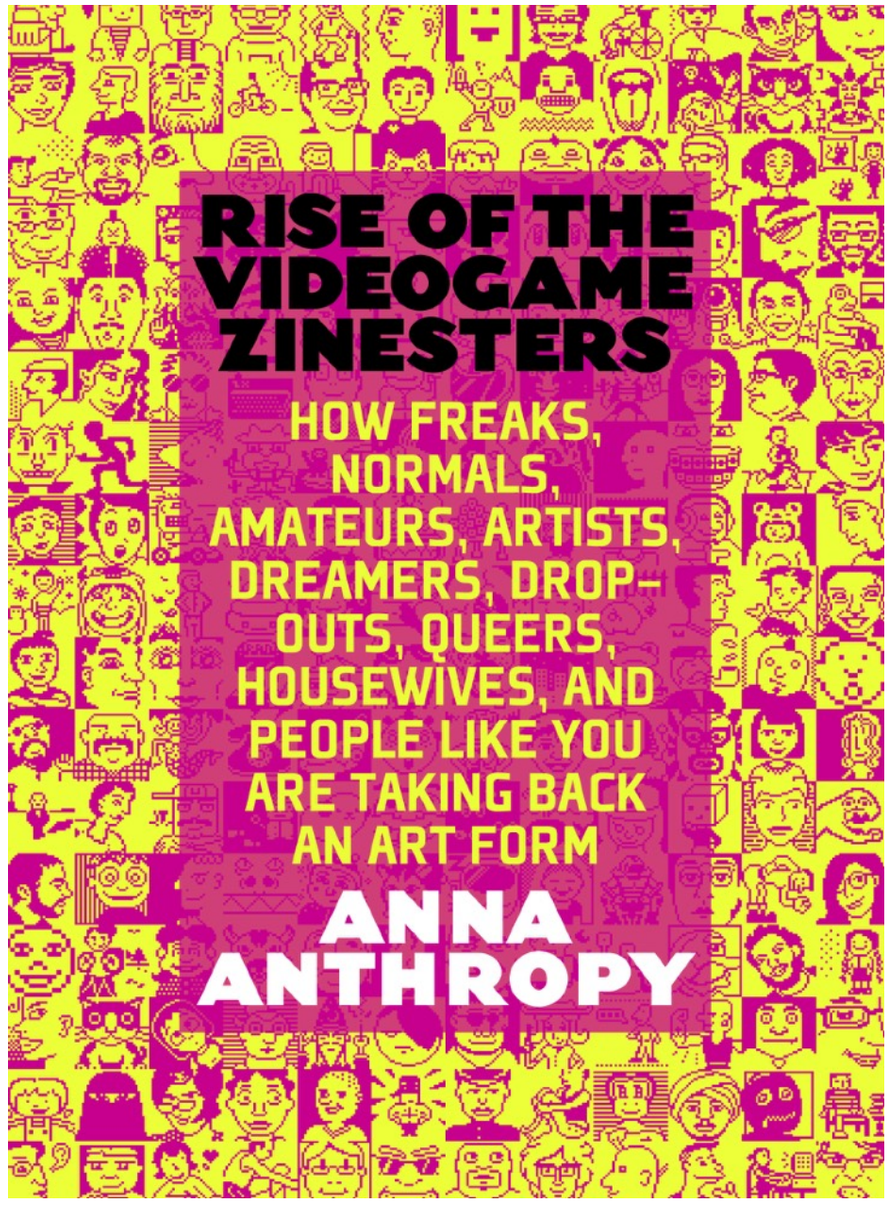
Why make a programming language?

- “DSLs” — “Domain Specific Languages”
- Little languages embedded inside other languages
    - The host language does all the real work, the DSL does exactly one thing
- Configuration as code

This was the talk I wanted to give and they told me I could

“Queerness in games”

- PL design is about taking control of your own destiny
- PL design is personal!



trashymctrashboy
@moshboy

Following

today you could make a bad videogame or if videogames aren't your thing just create something bad in any form because filling the world with bad things makes it infinitely more beautiful

5:53 PM - 30 Apr 2018

“Games”

- DSLs are unusually useful in gamedev
  - “Scripts”
- PLs mimic the structure of games
  - Logic systems
  - Logic puzzles
  - Have you played Exapunks yet it’s good like it’s really really good and I spent all last month playing it and the first line of the game is spoken by an enby which just meant a lot to me, like, I’m sorry I’m just really excited

Exapunks, Zachtronics, 2018

Demo

You can find the code used in the talk at
https://mcclure.github.io/qgcon-2018/

Why make a programming language?

When we write code, we are restructuring our thoughts to fit the language of the computer.
When we design our own languages, we rewire the computer to match the structure of our thoughts.

This is a song

↓

```
(P t--3 0 p8x p13x p0x p8x p13x)
(R t--2 0 t--3 p0 x p8 x p13 x)
R P -2 R P +4 R P -2
```

https://www.youtube.com/watch?v=bJCk2IcAwVE

Some tips

- Keep it simple
- Design your syntax around what’s easy to parse
- Python style indentation blocks are easier than { }s
- “JSON as code”— you might not need a parser!

Some tips

- Leverage your host language!
    - GC, nonlocal return (try/catch), tail calls, dynamic types
- …don’t trust your host language too much

Next steps: A stack

- In the demo, the “parser state” was global
- Instead keep it in a reusable object, and push/pop instances of this object on a “stack" array

```
class StackFrame {
    constructor(line, pos, scope) {
        this.line = line
        this.pos = pos
        this.scope = {}
        for (x in scope)
            this.scope[x] = scope[x]
    }
}
```

- Parse with loops, not recursion (so halt/resume is easy)

Next steps: PEG parsers

- They are everywhere
- There is one for your favorite programming language
- Just download it
- Based on “grammar syntax”
  - A = B | C A
    B = “X”
    C = “O”
  - Matches OOOOOOOOOOOOOX

```
// https://pegjs.org/online

Program "program"
  = EOL? _ head:Statement tail:(EOL _ Statement)* EOL? { return [head].concat(tail.map((x) => x[2])) }

Statement "statement"
  = "if" _ cond:Exp _ run:Statement { return ["if", cond, run] }
  / name:Var _ op:("="/"+="/"-=") _ val:Exp { return [op, name, val] }
  / Call / Label
  / "go" _ name:Var { return ["go", name] }
  / Var { return [text()] }

CondOp "conditional operator" = ([><] "="?) { return text() }

Exp "expression"
  = a:Cond _ op:CondOp _ b:Exp   { return [op, a, b] }
  / Cond

Cond "expression"
  = a:Plus _ op:[+-] _ b:Cond    { return [op, a, b] }
  / Plus

Plus "expression"
  = a:BaseExp _ op:[*/] _ b:Plus { return [op, a, b] }
  / BaseExp

BaseExp "expression" = Num / Call / Var / "(" _ e:Exp _ ")" { return e }

Call "function call"
  = f:Var _ "(" _ head:Exp _ tail:("," _ Exp _)* ")" { return [f, [head].concat(tail.map((x) => x[2]))] }

Var "variable" = ([a-z]i+) { return text() }

Label "label"  = ([a-z]i+) ":" { return [text()] }

Num "number" = "-"? [0-9]+ { return parseInt(text(), 10); }

EOL "whitespace" = (_ [\r\n])+

_ "whitespace" = [ \t]*
```

```
// https://pegjs.org/online

Statement “statement"          // A statement is
  = name:Var _                 // A name,
    op:("="/"+="/"-=") _       // then an operator,
    val:Expression           // then an expression.
      // When you see one, turn it into
      // an array, with the operator first.
      { return [op, name, val] }

/ Expression  // A statement can also be an
              // expression by itself.
              // A “/“ means “or”.
```

2 Test the generated parser with some input

```
target = playerx
if (firing) go flee
go ready
flee:
    if (x < playerx) target += 300
    if (playerx > x) target -= 300
ready:
        test = 3 * 4 * 4 + 3 * 2 + 4
        if (target < x)  speed -= acc
        if (target > x)  speed += acc
        if (speed > 30)  speed = 30
        if (speed < -30) speed = -30
        x += speed
        wait
```

```
[
    [ "=", "target", "playerx” ],
    [ "if", "firing", ["go", "flee"] ],
    [ "go", "ready" ],
    [ "flee:" ],
    [ "if", ["<", "x", "playerx”], ["+=", "target", 300] ],
    [ "if", [">", "playerx", "x”], ["-=", "target", 300] ],
    [ "ready:" ],
    [ "if", ["<", "target", "x"], ["-=", "speed", "acc" ] ],
    [ "if", [">", "target", "x”], ["+=", "speed", "acc" ] ],
    [ "if", [">", "speed", 30], ["=", "speed", 30] ],
    [ "if”, ["<", "speed", -30], ["=", "speed", -30 ] ],
    [ "+=", "x", "speed" ],
    ["wait”]
]
```

2 Test the generated parser with some input

```
        test = 3 * 4 * 4 + 3 * 2 + 4

anothertest = (3 + 4) * func(1, 2*3, 3)
```

```
[
   ["=", "test",
      ["+", ["*", 3, ["*", 4, 4]], ["+", ["*", 3, 2], 4]]],
   ["=", "anothertest",
      ["*", ["+", 3, 4],
         ["func",
            [
               1,
               ["*", 2, 3],
               3
            ]
         ]
      ]
   ]
]
```

Next steps: Compiling LLVM

- Construct instructions as data structures in memory,
  save to .exe, save to WebAssembly, create a .o and link it with C

- This is equivalent to `increment: value = value + 1`

```
local increment = LLVM.LLVMAppendBasicBlock(mainFunction, "Increment")
LLVM.LLVMPositionBuilderAtEnd(builder.llvm.builder, increment)

local valueDeref = LLVM.LLVMBuildLoad(builder.llvm.builder, value, "ValueDeref")

-- *value = *value + 1
local addResult = LLVM.LLVMBuildAdd(builder.llvm.builder, valueDeref,
    LLVM.LLVMConstInt(int32Type, 1, false), "AddResult")
LLVM.LLVMBuildStore(builder.llvm.builder, addResult, value)
```

- I wrote a blog series titled “No Compiler”
  https://msm.runhello.com/p/1003
  https://msm.runhello.com/p/1013
  https://msm.runhello.com/p/1022

Next steps: Compiling to C#

- System.Reflection.Emit is magic
  - Construct bytecode in memory
  - Save byte code to a .dll on disk
  - Feed .dll to Unity
  - https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.assemblybuilder?view=netframework-4.7.2

Next steps: Compiling to C#

- Okay, it's a little ugly

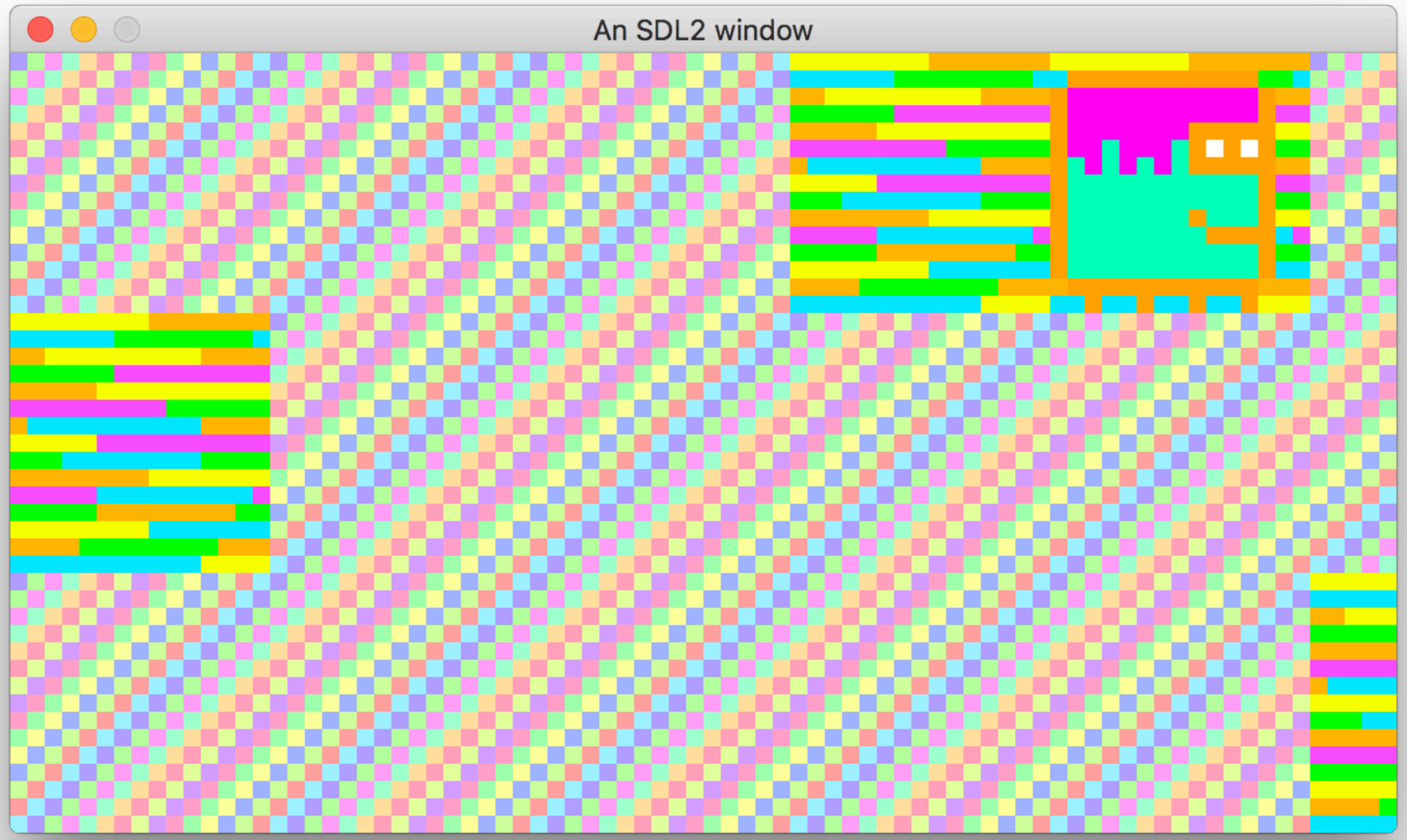
```
// Add a private field of type int (Int32).
FieldBuilder fbNumber = tb.DefineField(
    "m_number",
    typeof(int),
    FieldAttributes.Private);

// Define a constructor that takes an integer argument and
// stores it in the private field.
Type[] parameterTypes = { typeof(int) };
ConstructorBuilder ctor1 = tb.DefineConstructor(
    MethodAttributes.Public,
    CallingConventions.Standard,
    parameterTypes);

ILGenerator ctor1IL = ctor1.GetILGenerator();
// For a constructor, argument zero is a reference to the new
// instance. Push it on the stack before calling the base
// class constructor. Specify the default constructor of the
// base class (System.Object) by passing an empty array of
// types (Type.EmptyTypes) to GetConstructor.
ctor1IL.Emit(OpCodes.Ldarg_0);
ctor1IL.Emit(OpCodes.Call,
    typeof(object).GetConstructor(Type.EmptyTypes));
// Push the instance on the stack before pushing the argument
// that is to be assigned to the private field m_number.
ctor1IL.Emit(OpCodes.Ldarg_0);
ctor1IL.Emit(OpCodes.Ldarg_1);
ctor1IL.Emit(OpCodes.Stfld, fbNumber);
ctor1IL.Emit(OpCodes.Ret);
```

Escape velocity

- What if you want to make a Real Language?
    - Put it off as long as you can
    - Build it on top of an existing system
    - Care about community management
    - Beware of quicksand
        - Type systems are dangerous
        - DON'T WRITE YOUR OWN PACKAGE MANAGER

Escape velocity

- What if you want to make a Real Language?
  - Make sure you’re doing it for you



https://vine.co/v/e2Zqhe91tbF

More of my stuff if you care

Emily: https://emilylang.org/
(Check out the YouTube video)

Games: https://mermaid.industries
(Check out the link to Run Hello)

Twitter (ugh): @mcclure111, @MermaidVR
Mastodon: @mcc@mastodon.social

Questions?