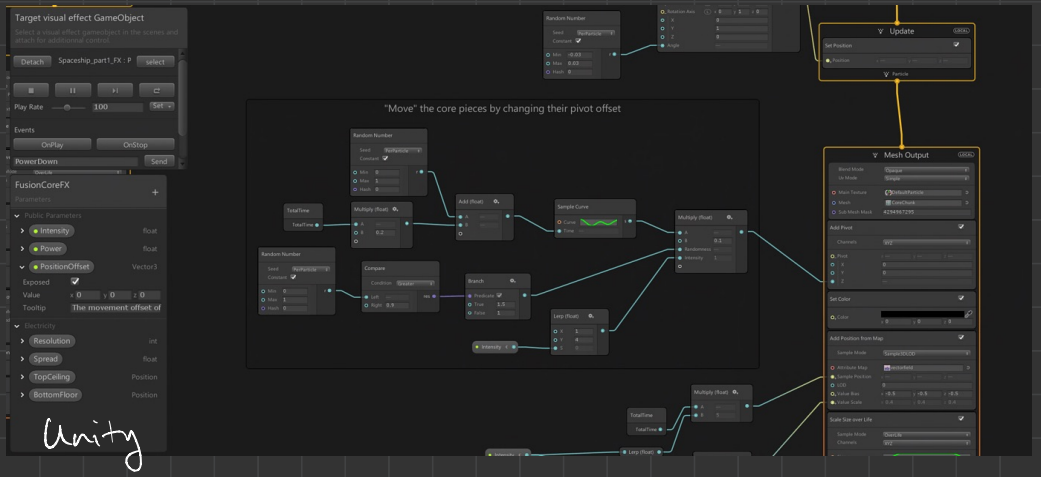
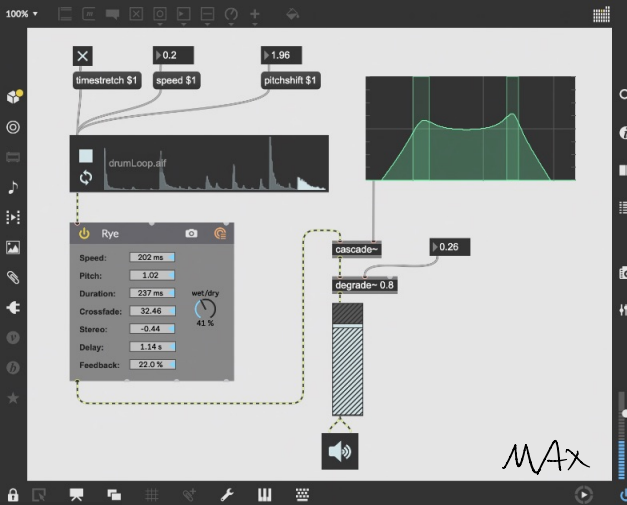
Babka is You

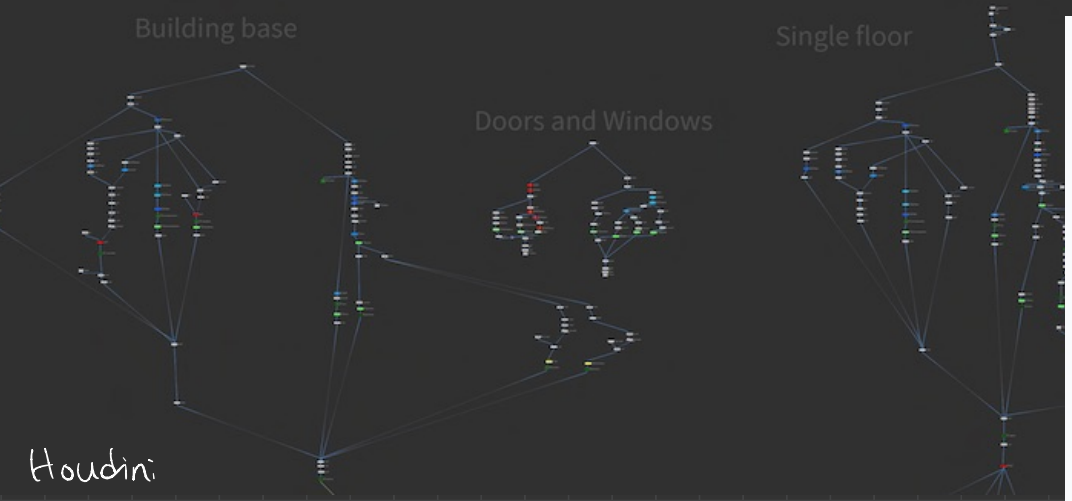
Math is fun!

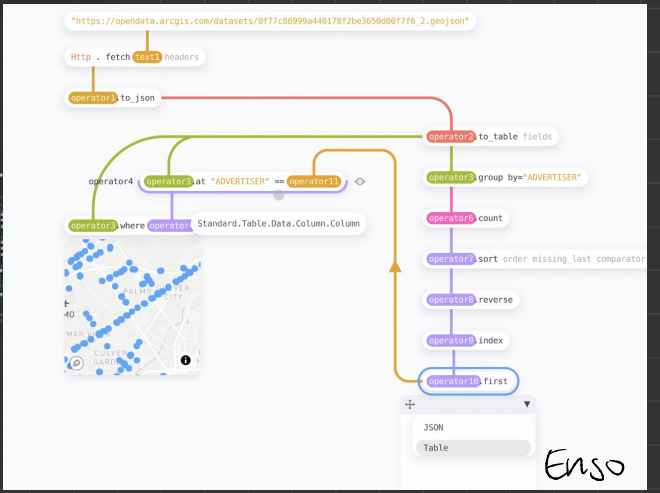
(sometimes)

I Can Talk
About Things
besides
Data Pipelines
and
Positive Pooping

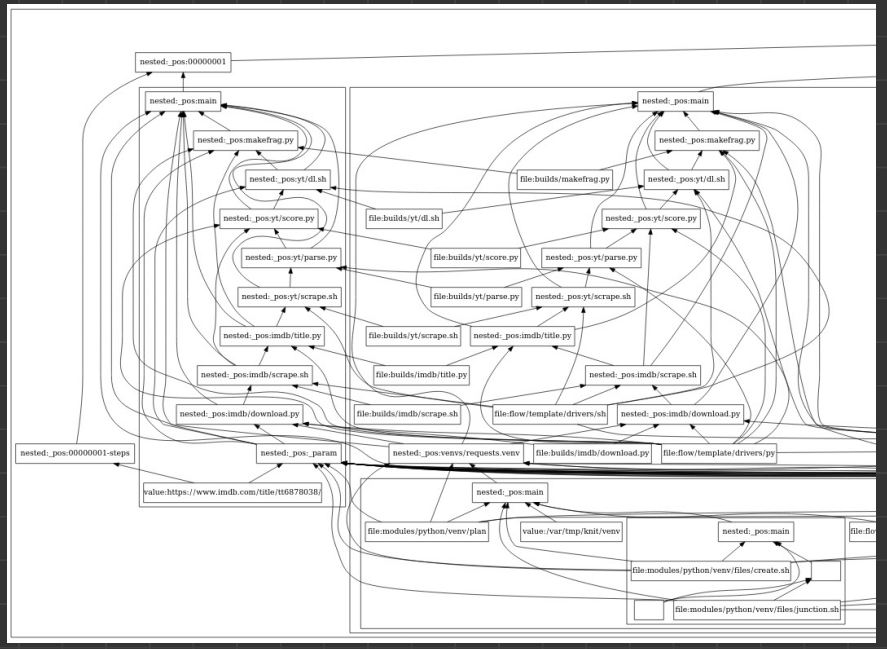

Target visual effect GameObject
"Move" the core pieces by changing their pivot offset
FusionCoreFX
Unity

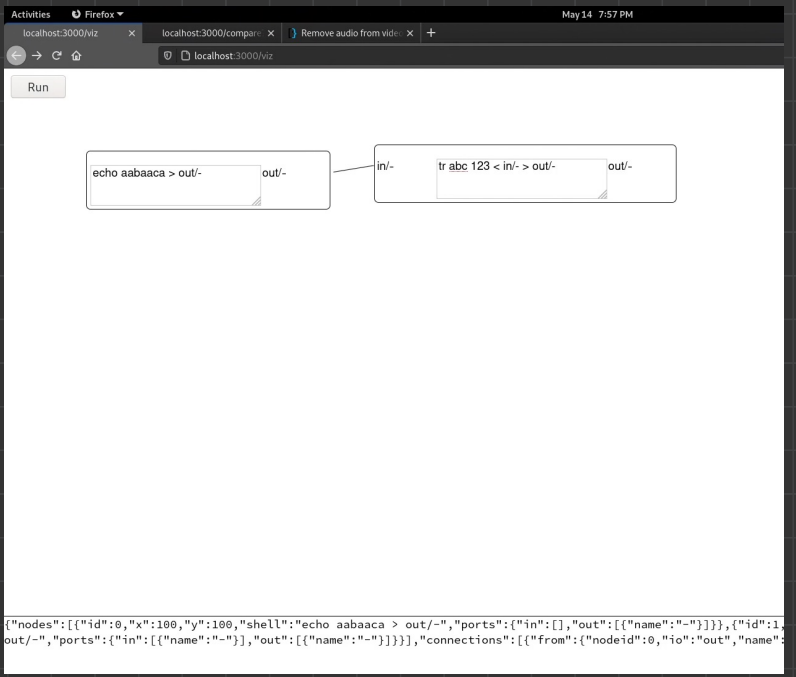

timestretch $1
speed $1
pitchshift $1
drumLoop.aif
Rye
cascade~
degrade~ 0.8
MAx


Building base
Doors and Windows
Single floor
Houdini

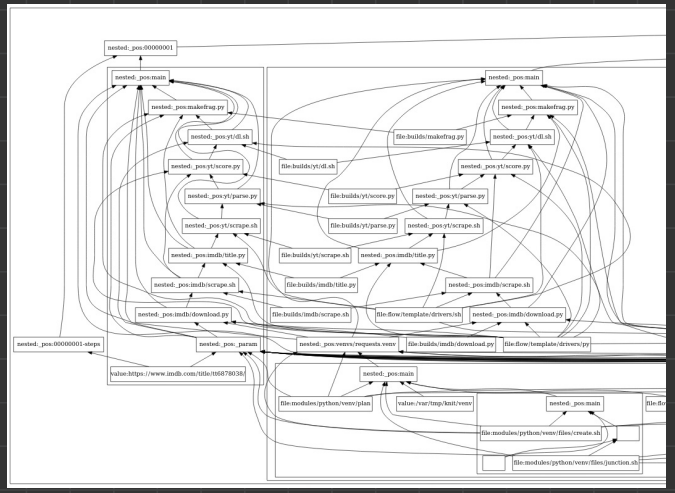

Enso

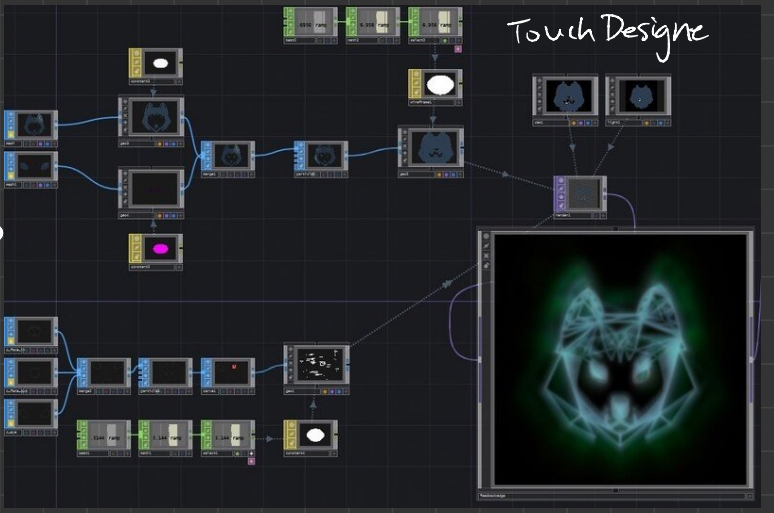
Ok, some data pipelines talk





How hard could it be?





Entity	unique identifier
Component	data
System	behaviors and interactions

```
class ECS {
  constructor() {
    this.maxEntity = 0;
    this.components = {};
  }

  addEntity(...components) {
    const entity = ++this.maxEntity;
    for (const c of components) {
      this.attach(entity, c);
    }
    return entity;
  }

  attach(entity, component) {
    (this.components[component.constructor.name] ||= {})[entity] = component;
  }
}
```

integer identifier

add entities

attach components

```
{
  maxEntity: 2
  components: {
    ComponentFoo: {
      1: ComponentFoo()
    }
  }
}
```

```
class ComponentBounds {
  constructor(x, y, width, height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
  }
}

class ComponentWhiteBox {}
```

components


class ComponentBounds {
constructor(x, y, width, height) {
this.x = x;
this.y = y;
this.width = width;
this.height = height;
}
}
class ComponentWhiteBox {}
Components
System
class SystemRender {
constructor(world, canvas) {
this.world = world;
this.ctx = canvas.getContext('2d');
}
render() {
const { width, height } = this.ctx.canvas;
this.ctx.clearRect(0, 0, width, height);
for (const entity in this.world.components.ComponentWhiteBox || []) {
const { x, y, width, height } = this.world.components.ComponentBounds[entity];
this.ctx.fillStyle = 'white';
this.ctx.fillRect(x, y, width, height);
}
}
}


```
window.onload = function() {
  world = new ECS();
  const canvas = document.getElementById('game');
  const render = new SystemRender(world, canvas);
  world.addEntity(
    new ComponentBounds(100, 100, 80, 50),
    new ComponentWhiteBox(),
  );
  render.render();
};
```

```
class ComponentBounds {
  constructor(x, y, width, height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
  }
}

class ComponentWhiteBox {}
```

components

system

```
class SystemRender {
  constructor(world, canvas) {
    this.world = world;
    this.ctx = canvas.getContext('2d');
  }

  render() {
    const { width, height } = this.ctx.canvas;
    this.ctx.clearRect(0, 0, width, height);
    for (const entity in this.world.components.ComponentWhiteBox || []) {
      const { x, y, width, height } = this.world.components.ComponentBounds[entity];
      this.ctx.fillStyle = 'white';
      this.ctx.fillRect(x, y, width, height);
    }
  }
}
```

components

entity

system

```
window.onload = function() {
  world = new ECS();
  const canvas = document.getElementById('game');
  const render = new SystemRender(world, canvas);
  world.addEntity(
    new ComponentBounds(100, 100, 80, 50),
    new ComponentWhiteBox(),
  );
  render.render();
};
```

I.I

~60 LOC

```
class SystemInput {
  constructor(world, canvas) {
    this.world = world;
    this.canvas = canvas;
    this.transition('default');
    canvas.addEventListener('mousedown', this.mousedown.bind(this));
    canvas.addEventListener('mousemove', this.mousemove.bind(this));
    canvas.addEventListener('mouseup', this.mouseup.bind(this));
  }
```

```
  click(x, y, entity) {
    this.world.addEntity(
      new ComponentBounds(x, y, 80, 50),
      new ComponentWhiteBox(),
      new ComponentDrag(),
    );
  }

  drag(dx, dy, entity) {
    if (entity in this.world.components.ComponentDrag) {
      const bounds = this.world.components.ComponentBounds[entity];
      bounds.x += dx;
      bounds.y += dy;
    }
  }
```

click ⇒ add draggable white box at (x,y)

drag ⇒ adjust (x,y) position

```
  hit(x, y) {
    for (const entity in this.world.components.ComponentDrag || []) {
      const bounds = this.world.components.ComponentBounds[entity];
      if (bounds && bounds.x < x && x < bounds.x + bounds.width &&
          bounds.y < y && y < bounds.y + bounds.height) {
        return entity;
      }
    }
  }
```

hit test which entity is under mouse pointer

I.II

~180 LOC

White boxes are boring

```
class ComponentSprite {
  constructor(img, scale = 1) {
    this.canvas = document.createElement('canvas');
    this.canvas.width = img.width * scale;
    this.canvas.height = img.height * scale;
    const ctx = this.canvas.getContext('2d');
    ctx.imageSmoothingEnabled = false;
    ctx.drawImage(img, 0, 0, this.canvas.width, this.canvas.height);
  }
}
```

load sprite image data

```
class SystemSpriteRender
  render() {
    this.dirty = false;
    const { width, height } = this.ctx.canvas;
    this.ctx.clearRect(0, 0, width, height);
    this.world.signal('vblank', { width, height });

    for (const [entity, sprite] of this.world.queryComponent(ComponentSprite)) {
      const { x, y } = this.world.cast(entity, ComponentBounds);
      this.ctx.drawImage(sprite.canvas, x, y);
      this.world.signal('blit', entity);
    }

    this.world.signal('scanout');
  }
}
```

display sprite image

```
let img = document.getElementById('flag');
world.addEntity(
  new ComponentBounds(100, 100, 840, 800),
  new ComponentSprite(img, 40),
  new ComponentDrag(),
);
```

II.I

~270 LOC

BTW getting some help

event handling

```
class World extends ECS {
  constructor() {
    super();
    this.listeners = {};
  }

  listen(type, listener) {
    (this.listeners[type] ||= []).push(listener);
  }

  signal(type, data) {
    for (const listener of this.listeners[type] || []) {
      listener(data);
    }
  }

  detach(entity, componentClass) {
    delete this.components[componentClass.name]?.[entity];
  }

  destroy(entity) {
    for (const c of Object.values(this.components)) {
      delete c[entity];
    }
  }

  queryComponent(componentClass) {
    const components = this.components[componentClass.name] || {};
    return Object.entries(components);
  }

  cast(entity, componentClass) {
    return this.components[componentClass.name]?.[entity];
  }
}
```

Time to talk about hit testing

```
hit(x, y) {
  for (const entity in this.world.components.ComponentDrag || []) {
    const bounds = this.world.components.ComponentBounds[entity];
    if (bounds && bounds.x < x && x < bounds.x + bounds.width &&
        bounds.y < y && y < bounds.y + bounds.height) {
      return entity;
    }
  }
}
```

slow!

what if multiple entities overlap?

```
class SystemHitInput extends SystemInput {
  constructor(world, canvas, hitcanvas) {
    super(world, canvas);
    this.hitcanvas = hitcanvas;
    world.listen('vblank', this.vblank.bind(this));
    world.listen('blit', this.blit.bind(this));
    world.listen('scanout', this.scanout.bind(this));
  }
```

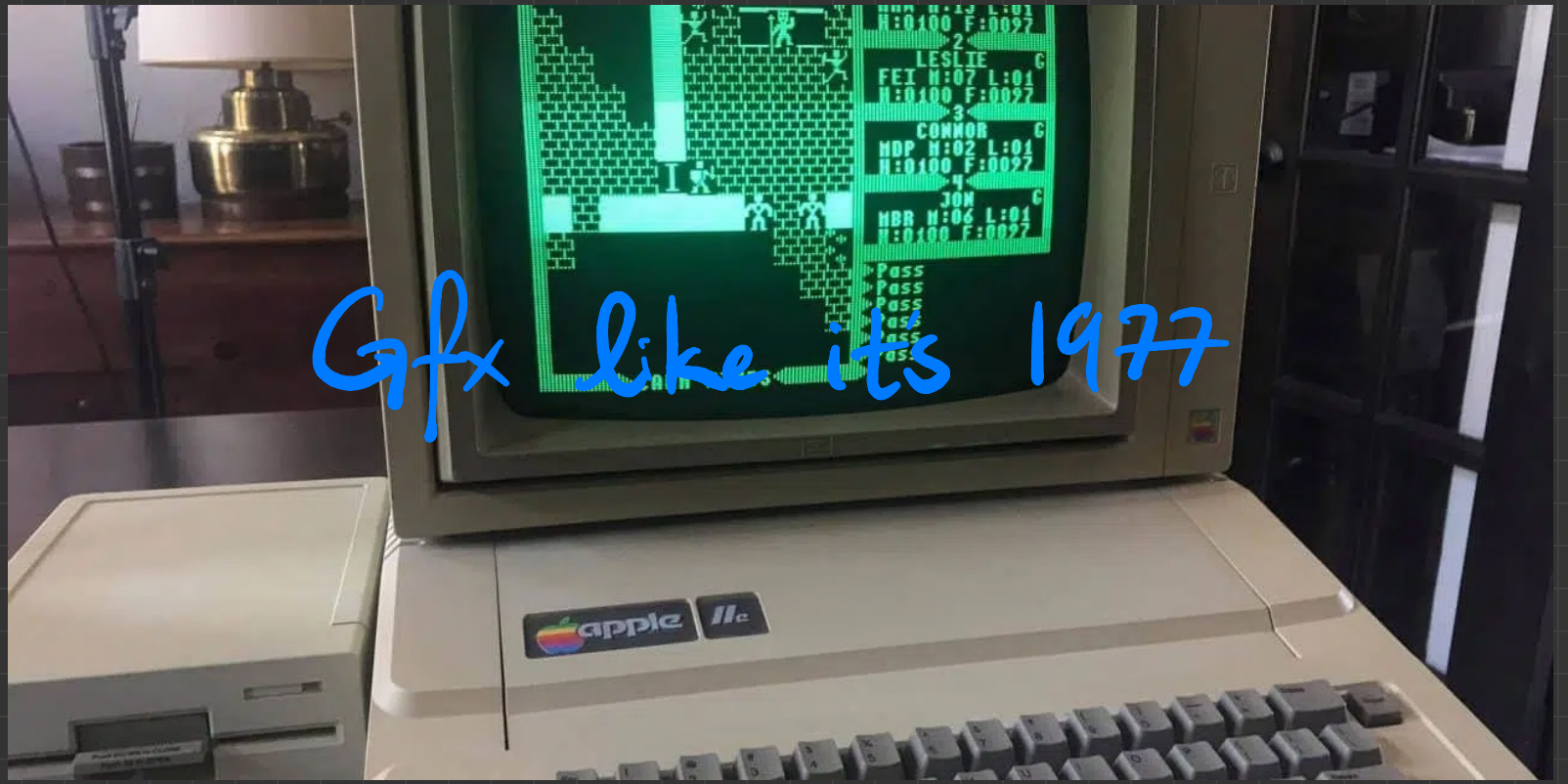
} listen for rendering events...

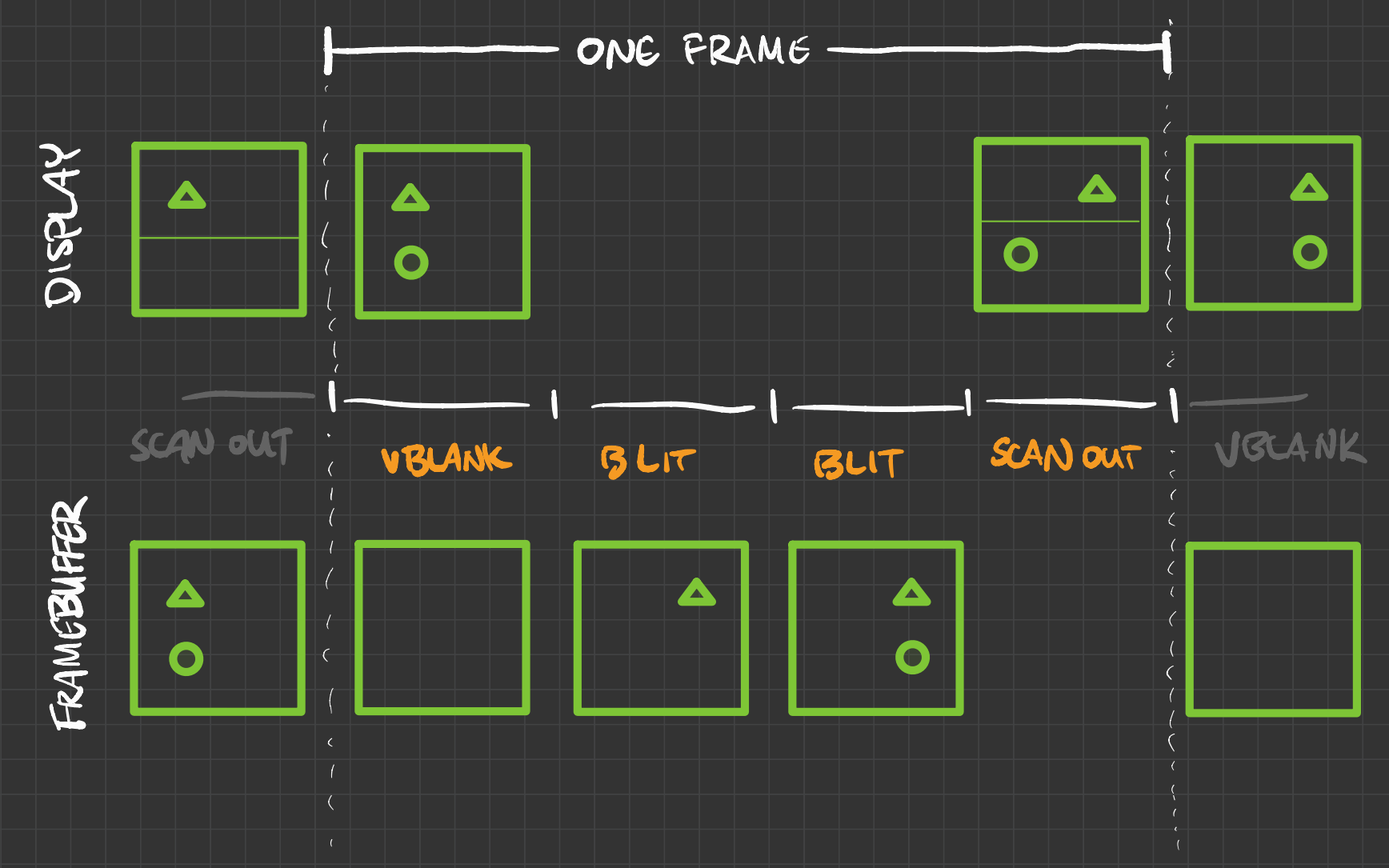
```
  vblank({ width, height }) {
    this.hitcanvas.width = width;
    this.hitcanvas.height = height;
    this.hitctx = this.hitcanvas.getContext('2d', { alpha: false });
    this.hitctx.clearRect(0, 0, width, height);
    // Drawing must be pixel perfect (no antialiasing) for proper hit detection.
  }

  blit(entity) {
    const { x, y, width, height } = this.world.cast(entity, ComponentBounds);
    this.hitctx.fillStyle = this.swizzle(entity);
    this.hitctx.fillRect(Math.floor(x), Math.floor(y), Math.ceil(width), Math.ceil(height));
  }

  scanout() {
    this.hitmap = this.hitctx.getImageData(0, 0, this.hitcanvas.width, this.hitcanvas.height);
  }
```

... to draw a second hitmap buffer

II. II

~340 LoC


Gfx like it's 1977
H:0100 F:0097
2
LESLIE C
FEI H:07 L:01
H:0100 F:0027
3
CONNOR C
MDP H:02 L:01
H:0100 F:0027
4
JON C
MBR H:06 L:01
H:0100 F:0027
Pass
Pass
Pass
apple IIe

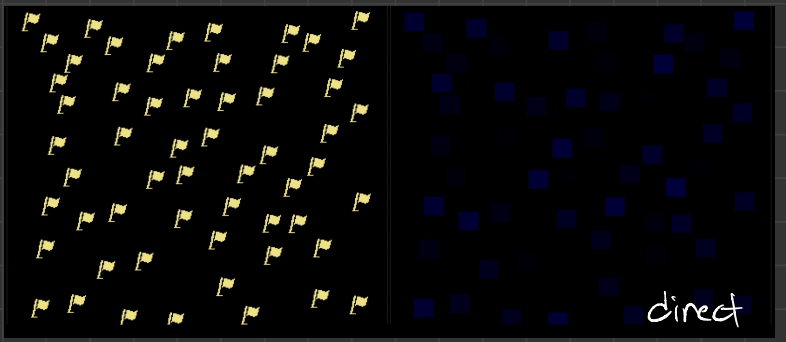

ONE FRAME
DISPLAY
SCAN OUT
VBLANK
BLIT
BLIT
SCAN OUT
VBLANK
FRAMEBUFFER

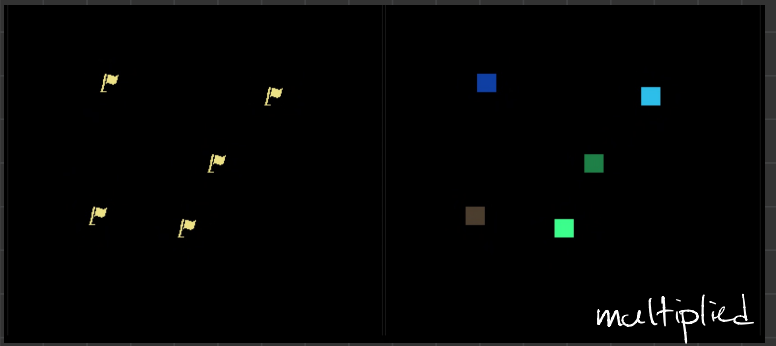
What the swizzle?

"Modular multiplicative inverse"

```
swizzle(entity) {
  // Encode up to 24 bits in RGB color. Multiply for visually distinct colors.
  const swizzled = (entity * 999331) % (1 << 24);
  return '#' + (swizzled).toString(16).padStart(6, 0);
}

hit(x, y) {
  const offset = (y * this.hitcanvas.width + x) * 4;
  let swizzled = 0;
  for (let i = 0; i < 3; i++ ) {
    swizzled <<= 8;
    swizzled += this.hitmap.data[offset + i];
  }
  if (!swizzled)
    return undefined;
  // Multiplicative inverse of swizzle() multiple.
  return (swizzled * 8055819) % (1 << 24)
}
```

Math is 🤩






BABA IS YOU
z

entity-component-system?

```
class ComponentYou {}
class ComponentPush {}
class ComponentStop {}
class ComponentSink {}
class ComponentWin {}
```

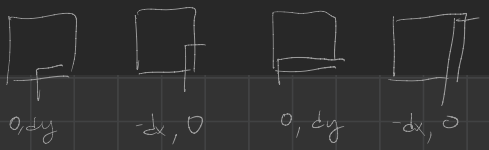
"interaction" components

```
class ComponentYou {}
class ComponentPush {}
class ComponentStop {}
class ComponentSink {}
class ComponentWin {}
```

"interaction" components

```
ComponentBounds.prototype.overlaps = function(o) {
  let dx;
  let dy;

  if (this.x <= o.x && o.x < this.x + this.width) {
    dx = this.x + this.width - o.x;
  }
  if (this.x <= o.x + o.width && o.x + o.width < this.x + this.width) {
    dx = dx ? 99 : this.x - o.x - o.width;
  }

  if (this.y <= o.y && o.y < this.y + this.height) {
    dy = this.y + this.height - o.y;
  }
  if (this.y <= o.y + o.height && o.y + o.height < this.y + this.height) {
    dy = dy ? 99 : this.y - o.y - o.height;
  }

  if (!dx || !dy) return null;
  if (Math.abs(dx) < Math.abs(dy)) {
    return [dx, 0];
  } else {
    return [0, dy];
  }
}
```

math is 😫



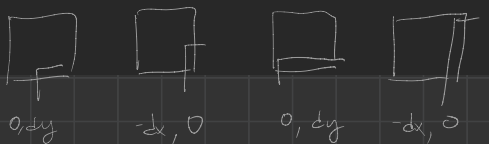
```
class ComponentYou {}
class ComponentPush {}
class ComponentStop {}
class ComponentSink {}
class ComponentWin {}
```

"interaction" components

```
class SystemGameInput extends SystemHitInput
  drag(dx, dy, entity) {
    for (const [entity, ] of this.world.queryComponent(ComponentYou)) {
      const bounds = this.world.cast(entity, ComponentBounds);
      bounds.x += dx;
      bounds.y += dy;
      this.collide(entity);

      for (const [collision, ] of this.world.queryComponent(ComponentWin)) {
        if (this.world.cast(collision, ComponentBounds).overlaps(bounds)) {
          this.mouseup(event);
          this.world.signal('recomposite');
          if (this.won) return;  // hacky debounce
          this.won = true;
          this.world.signal('win');
          return;
        }
      }
    }
    this.world.signal('recomposite');
  }
```

```
ComponentBounds.prototype.overlaps = function(o) {
  let dx;
  let dy;

  if (this.x <= o.x && o.x < this.x + this.width) {
    dx = this.x + this.width - o.x;
  }
  if (this.x <= o.x + o.width && o.x + o.width < this.x + this.width) {
    dx = dx ? 99 : this.x - o.x - o.width;
  }

  if (this.y <= o.y && o.y < this.y + this.height) {
    dy = this.y + this.height - o.y;
  }
  if (this.y <= o.y + o.height && o.y + o.height < this.y + this.height) {
    dy = dy ? 99 : this.y - o.y - o.height;
  }

  if (!dx || !dy) return null;
  if (Math.abs(dx) < Math.abs(dy)) {
    return [dx, 0];
  } else {
    return [0, dy];
  }
}
```

math is 😫




```
  collide(entity) {
    const bounds = this.world.cast(entity, ComponentBounds);
    let adjust = [0, 0];

    for (const [collision, ] of this.world.queryComponent(ComponentSink)) {
      if (this.world.cast(collision, ComponentBounds).overlaps(bounds)) {
        this.world.destroy(entity);
        this.world.destroy(collision);
        return adjust;
      }
    }

    for (const [collision, ] of this.world.queryComponent(ComponentStop)) {
      const overlap = this.world.cast(collision, ComponentBounds).overlaps(bounds);
      if (overlap) {
        adjust[0] += overlap[0];
        adjust[1] += overlap[1];
      }
    }
    bounds.x += adjust[0];
    bounds.y += adjust[1];
```

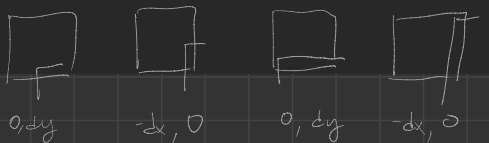
```
class ComponentYou {}
class ComponentPush {}
class ComponentStop {}
class ComponentSink {}
class ComponentWin {}
```

"interaction" components

```
class SystemGameInput extends SystemHitInput
  drag(dx, dy, entity) {
    for (const [entity, ] of this.world.queryComponent(ComponentYou)) {
      const bounds = this.world.cast(entity, ComponentBounds);
      bounds.x += dx;
      bounds.y += dy;
      this.collide(entity);

      for (const [collision, ] of this.world.queryComponent(ComponentWin)) {
        if (this.world.cast(collision, ComponentBounds).overlaps(bounds)) {
          this.mouseup(event);
          this.world.signal('recomposite');
          if (this.won) return;  // hacky debounce
          this.won = true;
          this.world.signal('win');
          return;
        }
      }
    }
    this.world.signal('recomposite');
  }

  collide(entity) {
    const bounds = this.world.cast(entity, ComponentBounds);
    let adjust = [0, 0];

    for (const [collision, ] of this.world.queryComponent(ComponentSink)) {
      if (this.world.cast(collision, ComponentBounds).overlaps(bounds)) {
        this.world.destroy(entity);
        this.world.destroy(collision);
        return adjust;
      }
    }

    for (const [collision, ] of this.world.queryComponent(ComponentStop)) {
      const overlap = this.world.cast(collision, ComponentBounds).overlaps(bounds);
      if (overlap) {
        adjust[0] += overlap[0];
        adjust[1] += overlap[1];
      }
    }
    bounds.x += adjust[0];
    bounds.y += adjust[1];
```

```
ComponentBounds.prototype.overlaps = function(o) {
  let dx;
  let dy;

  if (this.x <= o.x && o.x < this.x + this.width) {
    dx = this.x + this.width - o.x;
  }
  if (this.x <= o.x + o.width && o.x + o.width < this.x + this.width) {
    dx = dx ? 99 : this.x - o.x - o.width;
  }

  if (this.y <= o.y && o.y < this.y + this.height) {
    dy = this.y + this.height - o.y;
  }
  if (this.y <= o.y + o.height && o.y + o.height < this.y + this.height) {
    dy = dy ? 99 : this.y - o.y - o.height;
  }

  if (!dx || !dy) return null;
  if (Math.abs(dx) < Math.abs(dy)) {
    return [dx, 0];
  } else {
    return [0, dy];
  }
}
```

math is 😫



```
function addSprite(r, c, asset, ...components) {
  const scale = 3;
  const x = c * 24 * scale;
  const y = r * 24 * scale;

  const img = document.getElementById(asset);
  const entity = world.addEntity(
    new ComponentBounds(x, y, scale * img.width, scale * img.height),
    new ComponentSprite(img, scale),
    new ComponentHit(),
    ...components,
  );

  world.signal('recomposite');
  return entity;
}
```

assemblage

```
window.onload = function() {
  const canvas = document.getElementById('game');
  world = new World();
  new SystemSpriteRender(world, canvas);
  new SystemGameInput(world, canvas, document.getElementById('hit'));

  addSprite(2, 5, 'rock', new ComponentPush());
  addSprite(2, 9, 'wall', new ComponentStop());
  addSprite(6, 5, 'water', new ComponentSink());
  addSprite(6, 9, 'flag', new ComponentWin());
  addSprite(4, 7, 'baba', new ComponentYou());

  world.listen('win', () => alert('you win!'));
};
```

attach interaction components

III.I

~490 LOC

Dynamic game rules

```
class ComponentNoun {
  constructor(noun) {
    this.noun = noun;
  }
}

class ComponentIs {}

class ComponentVerb {
  constructor(componentClass) {
    this.componentClass = componentClass;
  }
}

class ComponentSubject {
  constructor(noun) {
    this.noun = noun;
  }
}
```

words

```
class ComponentNoun {
  constructor(noun) {
    this.noun = noun;
  }
}

class ComponentIs {}

class ComponentVerb {
  constructor(componentClass) {
    this.componentClass = componentClass;
  }
}

class ComponentSubject {
  constructor(noun) {
    this.noun = noun;
  }
}
```

words

remove interactions


```
class SystemBabaInput extends SystemGameInput {
  constructor(world, canvas, hitcanvas) {
    super(world, canvas, hitcanvas);
    this.rules = [];
    world.listen('scanout', this.attachDynamicComponents.bind(this));
  }

  attachDynamicComponents() {
    // Clear all rules
    for (const [entity, ] of this.world.queryComponent(ComponentSubject)) {
      this.world.detach(entity, ComponentYou);
      this.world.detach(entity, ComponentWin);
      this.world.detach(entity, ComponentStop);
      this.world.detach(entity, ComponentPush);
      this.world.detach(entity, ComponentSink);
    }
    this.rules = [];

    // Find active rules
    for (const [entity, ] of this.world.queryComponent(ComponentIs)) {
      const { x, y, width, height } = this.world.cast(entity, ComponentBounds);
      for (const [[x1, y1], [x2, y2]] of [[[x-1, y + height/2], [x+width, y + height/2]],
                                          [[x + width/2, y-1], [x + width/2, y+height]]]) {
        const before = this.hit(x1, y1);
        const after = this.hit(x2, y2);
        const noun = this.world.cast(before, ComponentNoun);
        const verb = this.world.cast(after, ComponentVerb);
        if (noun && verb) {
          this.rules.push([noun.noun, verb.componentClass]);
        }
      }
    }

    // Attach components (verbs) to entities (nouns)
    let debugString = [];
    for (const [wordNoun, componentClass] of this.rules) {
      for (const [entity, { noun }] of this.world.queryComponent(ComponentSubject)) {
        if (wordNoun == noun) {
          this.world.attach(entity, new componentClass());
        }
      }
      debugString.push(`${wordNoun} attached to ${componentClass.name}`);
    }
    document.getElementById('rules').innerHTML = debugString.join('<br>');
  }
```

```
class ComponentNoun {
  constructor(noun) {
    this.noun = noun;
  }
}

class ComponentIs {}

class ComponentVerb {
  constructor(componentClass) {
    this.componentClass = componentClass;
  }
}

class ComponentSubject {
  constructor(noun) {
    this.noun = noun;
  }
}
```

words


```
class SystemBabaInput extends SystemGameInput {
  constructor(world, canvas, hitcanvas) {
    super(world, canvas, hitcanvas);
    this.rules = [];
    world.listen('scanout', this.attachDynamicComponents.bind(this));
  }

  attachDynamicComponents() {
    // Clear all rules
    for (const [entity, ] of this.world.queryComponent(ComponentSubject)) {
      this.world.detach(entity, ComponentYou);
      this.world.detach(entity, ComponentWin);
      this.world.detach(entity, ComponentStop);
      this.world.detach(entity, ComponentPush);
      this.world.detach(entity, ComponentSink);
    }
    this.rules = [];

    // Find active rules
    for (const [entity, ] of this.world.queryComponent(ComponentIs)) {
      const { x, y, width, height } = this.world.cast(entity, ComponentBounds);
      for (const [[x1, y1], [x2, y2]] of [[[x-1, y + height/2], [x+width, y + height/2]],
                                          [[x + width/2, y-1], [x + width/2, y+height]]]) {
        const before = this.hit(x1, y1);
        const after = this.hit(x2, y2);
        const noun = this.world.cast(before, ComponentNoun);
        const verb = this.world.cast(after, ComponentVerb);
        if (noun && verb) {
          this.rules.push([noun.noun, verb.componentClass]);
        }
      }
    }

    // Attach components (verbs) to entities (nouns)
    let debugString = [];
    for (const [wordNoun, componentClass] of this.rules) {
      for (const [entity, { noun }] of this.world.queryComponent(ComponentSubject)) {
        if (wordNoun == noun) {
          this.world.attach(entity, new componentClass());
        }
      }
      debugString.push(`${wordNoun} attached to ${componentClass.name}`);
    }
    document.getElementById('rules').innerHTML = debugString.join('<br>');
  }
```

remove interactions

find noun is verb

```
class ComponentNoun {
  constructor(noun) {
    this.noun = noun;
  }
}

class ComponentIs {}

class ComponentVerb {
  constructor(componentClass) {
    this.componentClass = componentClass;
  }
}

class ComponentSubject {
  constructor(noun) {
    this.noun = noun;
  }
}
```

words

```
class SystemBabaInput extends SystemGameInput {
  constructor(world, canvas, hitcanvas) {
    super(world, canvas, hitcanvas);
    this.rules = [];
    world.listen('scanout', this.attachDynamicComponents.bind(this));
  }

  attachDynamicComponents() {
    // Clear all rules
    for (const [entity, ] of this.world.queryComponent(ComponentSubject)) {
      this.world.detach(entity, ComponentYou);
      this.world.detach(entity, ComponentWin);
      this.world.detach(entity, ComponentStop);
      this.world.detach(entity, ComponentPush);
      this.world.detach(entity, ComponentSink);
    }
    this.rules = [];

    // Find active rules
    for (const [entity, ] of this.world.queryComponent(ComponentIs)) {
      const { x, y, width, height } = this.world.cast(entity, ComponentBounds);
      for (const [[x1, y1], [x2, y2]] of [[[x-1, y + height/2], [x+width, y + height/2]],
                                          [[x + width/2, y-1], [x + width/2, y+height]]]) {
        const before = this.hit(x1, y1);
        const after = this.hit(x2, y2);
        const noun = this.world.cast(before, ComponentNoun);
        const verb = this.world.cast(after, ComponentVerb);
        if (noun && verb) {
          this.rules.push([noun.noun, verb.componentClass]);
        }
      }
    }

    // Attach components (verbs) to entities (nouns)
    let debugString = [];
    for (const [wordNoun, componentClass] of this.rules) {
      for (const [entity, { noun }] of this.world.queryComponent(ComponentSubject)) {
        if (wordNoun == noun) {
          this.world.attach(entity, new componentClass());
        }
      }
      debugString.push(`${wordNoun} attached to ${componentClass.name}`);
    }
    document.getElementById('rules').innerHTML = debugString.join('<br>');
  }
```

remove interactions

find noun is verb

attach verb components to noun subjects


```
window.onload = function() {
  const canvas = document.getElementById('game');
  world = new World();
  new SystemSpriteRender(world, canvas);
  new SystemBabaInput(world, canvas, document.getElementById('hit'));

  addSubject(2, 5, 'rock');
  addSubject(2, 9, 'wall');
  addSubject(6, 5, 'water');
  addSubject(6, 9, 'flag');
  addSubject(4, 7, 'baba');

  addNoun(0, 0, 'baba');
  addIs(0, 1);
  addVerb(0, 2, 'you');

  addNoun(3, 0, 'rock');
  addIs(3, 1);
  addVerb(3, 2, 'push');

  addNoun(4, 0, 'wall');
  addIs(4, 1);
  addVerb(4, 2, 'stop');

  addNoun(5, 0, 'water');
  addIs(5, 1);
  addVerb(5, 2, 'sink');

  addNoun(6, 0, 'flag');
  addIs(6, 1);
  addVerb(6, 2, 'win');

  world.listen('win', () => alert('dynamical!'));
};
```

Ⅲ.Ⅱ

~620ℓ

A

Joe Mou

Production

2021-11-03

https://github.com/jmou/babka-is-you

Flappy Eve

Setup
Draw the game world!

Game menus
Score calculation
Start a new game

Drawing
Player
Obstacles

Game Logic
Obstacles
Flapping the player
Scroll the world
Collision

Flappy Eve

When a player starts the game, we commit a `#world`, a `#player`, and some `#obstacles`. These will keep all of the essential state of the game. All of this information could have been stored on the world, but for clarity we break the important bits of state into objects that they effect.

- The `#world` tracks the distance the player has travelled, the current game screen, and the high score.
- The `#player` stores his current y position and (vertical) velocity.
- The `obstacles` have their (horizontal) offset and gap widths. We put distance on the world and only keep two obstacles; rather than moving the player through the world, we keep the player stationary and move the world past the player. When an obstacle goes off screen, we will wrap it around, update the placement of its gap, and continue on.

Setup

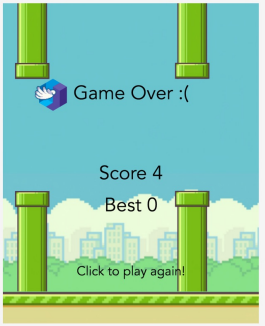
Add a flappy eve and a world for it to flap in:

```
commit
  [#player #self name: "eve" x: 25 y: 50 velocity: 0]
  [#world screen: "menu" frame: 0 distance: 0 best: 0 gravity: -0.061]
  [#obstacle gap: 35 offset: 0]
  [#obstacle gap: 35 offset: -1]
```

Next we draw the backdrop of the world. The player and obstacle will be drawn later based on their current state. Throughout the app we use resources from @bhauman's flappy bird demo in clojure. Since none of these things change over time, we commit them once when the player starts the game.

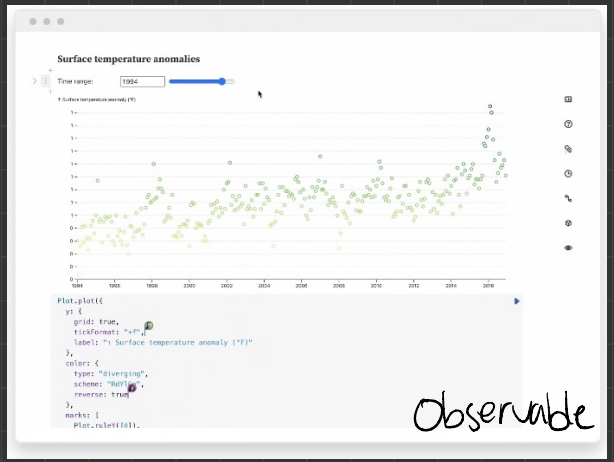
Draw the game world!

```
search
  world = [#world]

commit @browser
  world <- [#div style: [user-select: "none" -webkit-user-select: "none" -
moz-user-select: "none"] children:
    [#svg #game-window viewBox: "10 0 80 100", width: 480 children:
      [#rect x: 0 y: 0 width: 100 height: 53 fill: "rgb(112, 197, 206)" sort:
```

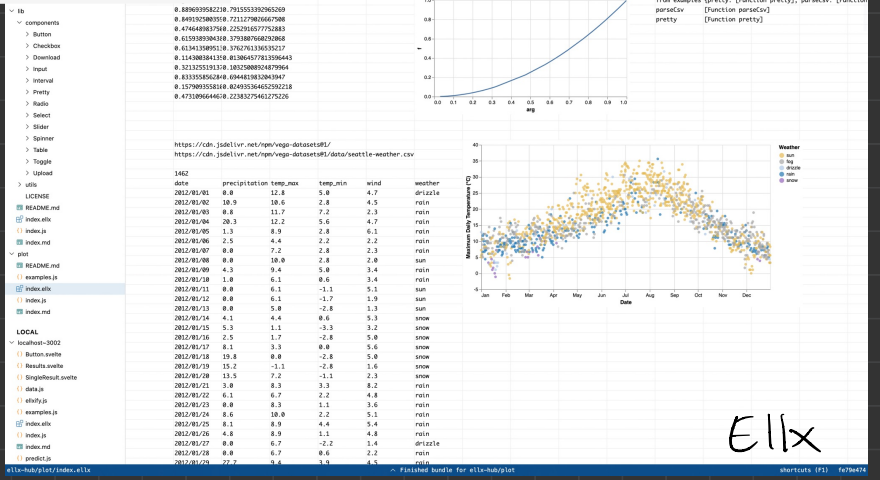


Other cool alternative programming languages

Eve



Observable



Elix