Why Design Your Own Levels When Your Computer Can Do It?

Updated Talk

Who Am I?

- Software Engineer.
- Adjunct Lecturer at CUNY.
- PhD student at the University of York.
- Functional Programming Enthusiast.

What am I not?

- I am not a game developer.
- I know very little about video game graphics.
- This is not a definitive guide to anything!

Why?

- About a year ago I wanted to try tackling one of my biggest blind spots in regards to programming: graphics.
- I also thought it might be cool to eventually make a proper game engine.
- Little did I know...

Graphics Are Horrible

- Working with graphics on modern systems is terrible.
  - DirectX is Windows-only
  - Metal is Mac/iOS-only.
  - OpenGL is in a bizarre nebulous state of quasi-support.
  - Vulkan makes you question every decision that led up to you using Vulkan.

Example of drawing a cube in Vulkan

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
#include <stdlib.h>
#include <stdio.h>

const uint32_t width = 800;
const uint32_t height = 600;

const float vertices[] = {
    -0.5f, -0.5f, -0.5f,
    -0.5f,  0.5f, -0.5f,
     0.5f, -0.5f, -0.5f,
     0.5f,  0.5f, -0.5f,
    -0.5f, -0.5f,  0.5f,
    -0.5f,  0.5f,  0.5f,
     0.5f, -0.5f,  0.5f,
     0.5f,  0.5f,  0.5f
};

const uint16_t indices[] = {
    0, 1, 2, 2, 1, 3,
    4, 5, 6, 6, 5, 7,
    0, 1, 4, 4, 1, 5,
    2, 3, 6, 6, 3, 7,
    0, 2, 4, 4, 2, 6,
    1, 3, 5, 5, 3, 7
};
```

```c
void initWindow(GLFWwindow** window)
{
    if (!glfwInit())
    {
        fprintf(stderr, "Failed to initialize GLFW\n");
        exit(EXIT_FAILURE);
    }

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

    *window = glfwCreateWindow(width, height, "Vulkan Cube", NULL, NULL);
    if (!(*window))
    {
        fprintf(stderr, "Failed to create GLFW window\n");
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
}
```

```
void cleanupWindow(GLFWwindow* window)
{
    glfwDestroyWindow(window);
    glfwTerminate();
}

void drawFrame(GLFWwindow* window, VkDevice device, VkCommandPool commandPool, VkQueue graphicsQueue, VkRenderPass renderPass, VkPipeline pipeline, VkBuffer vertexBuffer, VkBuffer indexBuffer, uint32_t indexCount)
{
    glfwPollEvents();

    uint32_t imageIndex;
    vkAcquireNextImageKHR(device, swapChain, UINT64_MAX, imageAvailableSemaphore, VK_NULL_HANDLE, &imageIndex);

    VkSubmitInfo submitInfo = {};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

    VkSemaphore waitSemaphores[] = { imageAvailableSemaphore };
    VkPipelineStageFlags waitStages[] = { VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT };
    submitInfo.waitSemaphoreCount = 1;
    submitInfo.pWaitSemaphores = waitSemaphores;
    submitInfo.pWaitDstStageMask = waitStages;

    VkCommandBuffer commandBuffers[] = { commandBuffer[imageIndex] };
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = commandBuffers;

    VkSemaphore signalSemaphores[] = { renderFinishedSemaphore };
    submitInfo.signalSemaphoreCount = 1;
    submitInfo.pSignalSemaphores = signalSemaphores;

    if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE) != VK_SUCCESS)
    {
        fprintf(stderr, "Failed to submit draw command buffer\n");
        exit(EXIT_FAILURE);
    }

    VkPresentInfoKHR presentInfo = {};
    presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;

    presentInfo.waitSemaphoreCount = 1;
    presentInfo.pWaitSemaphores = signalSemaphores;

    VkSwapchainKHR swapChains[] = { swapChain };
    presentInfo.swapchainCount = 1;
    presentInfo.pSwapchains = swapChains;
    presentInfo.pImageIndices = &imageIndex;

    if (vkQueuePresentKHR(presentQueue, &presentInfo) != VK_SUCCESS)
    {
        fprintf(stderr, "Failed to present image\n");
        exit(EXIT_FAILURE);
    }
}
```

```cpp
int main()
{
    GLFWwindow* window;

    initWindow(&window);

    VkInstance instance = createInstance();
    VkPhysicalDevice physicalDevice = selectPhysicalDevice(instance);
    VkDevice device = createLogicalDevice(physicalDevice);

    VkSwapchainKHR swapChain = createSwapChain(device, physicalDevice, window);
    VkRenderPass renderPass = createRenderPass(device, swapChain);
    VkPipeline pipeline = createGraphicsPipeline(device, renderPass);
    VkCommandPool commandPool = createCommandPool(device);
    VkCommandBuffer commandBuffer = createCommandBuffer(device, commandPool);

    VkBuffer vertexBuffer, indexBuffer;
    VkDeviceMemory vertexBufferMemory, indexBufferMemory;
    createVertexBuffer(device, physicalDevice, commandPool, graphicsQueue, &vertexBuffer, &vertexBufferMemory);
    createIndexBuffer(device, physicalDevice, commandPool, graphicsQueue, &indexBuffer, &indexBufferMemory);

    while (!glfwWindowShouldClose(window))
    {
        drawFrame(window, device, commandPool, graphicsQueue, renderPass, pipeline, vertexBuffer, indexBuffer, 36);
    }

    cleanupVertexBuffer(device, vertexBuffer, vertexBufferMemory);
    cleanupIndexBuffer(device, indexBuffer, indexBufferMemory);
    cleanupCommandBuffer(device, commandPool, commandBuffer);
    cleanupPipeline(device, pipeline);
    cleanupRenderPass(device, renderPass);
    cleanupSwapChain(device, swapChain);
    cleanupLogicalDevice(device);
    cleanupInstance(instance);

    cleanupWindow(window);

    return EXIT_SUCCESS;
}
```

NO!

- No
- No!
- NO!!

What are our other options?

- Unity or Unreal (or other pre-made game engine)
  - Nothing wrong with those, but I want to make my own stuff so I can be in charge of the design decisions.
  - Often limited in terms of languages and the like.
  - I'm an open source jerk.
- An abstraction layer

WebGL

- A JavaScript OpenGL runtime.
- Based on OpenGL ES

Potential perks to something like WebGL.

- Still somewhat low-level, giving us access to shaders and the like.
- Part of the web standard, meaning it's supported by nearly anything I care about.
  - Let the browser vendors deal with the Vulkan PTSD instead of me.

Disadvantage of WebGL?

- A bit slower than working with raw OpenGL or Vulkan.
- Confined to JavaScript (or are we?)

What is a shader?

- A shader is a little program that runs on the GPU.
- Fragment/Pixel Shader
  - Runs to determine pixel colors.
- Vertex Shaders
  - Runs to determine position of vertices in the 3D->2D space.

Example

```
const mat4 = glMatrix.mat4;

const vsSource = `
    attribute vec4 aVertexPosition;
    attribute vec4 aVertexColor;

    uniform mat4 uModelViewMatrix;
    uniform mat4 uProjectionMatrix;

    varying lowp vec4 vColor;

    void main(void) {
        gl_Position = uProjectionMatrix * uModelViewMatrix * aVertexPosition;
        vColor = aVertexColor;
    }
`;

const fsSource = `
    varying lowp vec4 vColor;

    void main(void) {
        gl_FragColor = vColor;
    }
`;
```

```
const cubeVertices = [
    // Front face
    -1.0, -1.0,  1.0,
     1.0, -1.0,  1.0,
     1.0,  1.0,  1.0,
    -1.0,  1.0,  1.0,

    ...
];

// Colors for the vertices
const cubeVertexColors = [
    [1.0, 1.0, 1.0, 1.0],
    ...
];

// Indices of vertices for each face
const cubeVertexIndices = [
    0,  1,  2,      0,  2,  3,
    ...
];
```

```
let gl;

function initWebGL(canvas) {
    gl = canvas.getContext('webgl');

    return gl;
}

function getShader(gl, type, source) {
    const shader = gl.createShader(type);

    gl.shaderSource(shader, source);
    gl.compileShader(shader);

    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        alert('An error occurred compiling the shaders: ' + gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
        return null;
    }

    return shader;
}

function initShaderProgram(gl, vsSource, fsSource) {
    const vertexShader = getShader(gl, gl.VERTEX_SHADER, vsSource);
    const fragmentShader = getShader(gl, gl.FRAGMENT_SHADER, fsSource);

    const shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert('Unable to initialize the shader program: ' + gl.getProgramInfoLog(shaderProgram));
        return null;
    }

    return shaderProgram;
}
```

```
function drawScene(gl, programInfo, buffers, deltaTime) {
    gl.clearColor(0.0, 0.0, 0.0, 1.0);  // Clear to black
    gl.clearDepth(1.0);                 // Clear everything
    gl.enable(gl.DEPTH_TEST);           // Enable depth testing
    gl.depthFunc(gl.LEQUAL);            // Near things obscure far things

    // Clear the canvas before we start drawing on it.
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    const fieldOfView = 45 * Math.PI / 180;   // in radians
    const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
    const zNear = 0.1;
    const zFar = 100.0;
    const projectionMatrix = mat4.create();

    mat4.perspective(projectionMatrix,
                     fieldOfView,
                     aspect,
                     zNear,
                     zFar);

    const modelViewMatrix = mat4.create();

    mat4.translate(modelViewMatrix,
                   modelViewMatrix,
                   [-0.0, 0.0, -6.0]);  // amount to translate

    mat4.rotate(modelViewMatrix,  // destination matrix
                modelViewMatrix,  // matrix to rotate
                cubeRotation,     // amount to rotate in radians
                [0, 0, 1]);       // axis to rotate around

    mat4.rotate(modelViewMatrix,  // destination matrix
                modelViewMatrix,  // matrix to rotate
                cubeRotation * .7,// amount to rotate in radians
                [0, 1, 0]);       // axis to rotate around

    // Position
    {
        const numComponents = 3;
        const type = gl.FLOAT;
        const normalize = false;
        const stride = 0;
        const offset = 0;
        gl.bindBuffer(gl.ARRAY_BUFFER, buffers.position);
        gl.vertexAttribPointer(
            programInfo.attribLocations.vertexPosition,
            numComponents,
            type,
            normalize,
            stride,
            offset);
        gl.enableVertexAttribArray(
            programInfo.attribLocations.vertexPosition);
    }

    // Color
    {
        const numComponents = 4;
        const type = gl.FLOAT;
        const normalize = false;
        const stride = 0;
        const offset = 0;
        gl.bindBuffer(gl.ARRAY_BUFFER, buffers.color);
        gl.vertexAttribPointer(
            programInfo.attribLocations.vertexColor,
            numComponents,
            type,
            normalize,
            stride,
            offset);
        gl.enableVertexAttribArray(
            programInfo.attribLocations.vertexColor);
    }

    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, buffers.indices);

    gl.useProgram(programInfo.program);

    gl.uniformMatrix4fv(
        programInfo.uniformLocations.projectionMatrix,
        false,
        projectionMatrix);
    gl.uniformMatrix4fv(
        programInfo.uniformLocations.modelViewMatrix,
        false,
        modelViewMatrix);

    {
        const vertexCount = 36;
        const type = gl.UNSIGNED_SHORT;
        const offset = 0;
        gl.drawElements(gl.TRIANGLES, vertexCount, type, offset);
    }

    cubeRotation += deltaTime;
}
```

```
let cubeRotation = 0.0;

function main() {
    const canvas = document.querySelector('#canvas');
    const gl = initWebGL(canvas);

    const shaderProgram = initShaderProgram(gl, vsSource, fsSource);

    const programInfo = {
        program: shaderProgram,
        attribLocations: {
            vertexPosition: gl.getAttribLocation(shaderProgram, 'aVertexPosition'),
            vertexColor: gl.getAttribLocation(shaderProgram, 'aVertexColor'),
        },
        uniformLocations: {
            projectionMatrix: gl.getUniformLocation(shaderProgram, 'uProjectionMatrix'),
            modelViewMatrix: gl.getUniformLocation(shaderProgram, 'uModelViewMatrix'),
        },
    };

    const buffers = initBuffers(gl);

    let then = 0;

    // Draw the scene repeatedly
    function render(now) {
        now *= 0.001;  // convert to seconds
        const deltaTime = now - then;
        then = now;

        drawScene(gl, programInfo, buffers, deltaTime);

        requestAnimationFrame(render);
    }
    requestAnimationFrame(render);
}

window.onload = main;
```

WebGL

- Better than Vulkan
- Still not fun.
- Very long.

p5.js

- Port/"reimagining" of the Java Processing library for JavaScript.
- Wrapper that can export/utilize WebGL.

Similar example

```
let angle = 0;

function setup() {
  createCanvas(710, 400, WEBGL);
}

function draw() {
  background(0);
  rotateX(angle);
  rotateY(angle * 0.3);
  rotateZ(angle * 1.2);

  // Draw the six faces of the cube, each with a different color.
  beginShape();
  fill(255, 0, 0); // Red face
  vertex(-50, -50, -50);
  vertex(50, -50, -50);
  vertex(50, 50, -50);
  vertex(-50, 50, -50);
  endShape(CLOSE);
  // repeat for the other five faces


  angle += 0.02;
}
```

p5.js

- This is much better
- ...
- But it's really imperative.
- Can we do better?

ClojureScript

- Port of Clojure that compiles to JavaScript.
- Fully embraces the JS ecosystem.
  - Lets you utilize Clojure's sexy data structures while also utilizing existing JavaScript libraries.

Why ClojureScript?

- It's Lisp and therefore cool.
- It compiles into JavaScript that's compatible down to very old Internet Explorer.
- The tooling for it is very good.

ShadowCLJS

- Development framework for ClojureScript.
- Handles dependency, automatic hot code reloading.
- Easily fascilitates interop between vanilla JavaScript and ClojureScript.

```clojure
(def angle (atom 0))

(defn draw-face [color coords]
  (js/fill color)
  (js/beginShape)
  (doseq [coord coords]
    (.vertex js/p5 (first coord) (second coord) (nth coord 2)))
  (js/endShape js/CLOSE))

(defn draw []
  (js/background 0)
  (js/rotateX @angle)
  (js/rotateY (* @angle 0.3))
  (js/rotateZ (* @angle 1.2))

  ;; Draw faces of the cube
  (let [coords [[-50 -50 -50] [50 -50 -50] [50 50 -50] [-50 50 -50]]]
    (draw-face [255 0 0] coords))  ;; red face
  ;; ... repeat for other faces

  (swap! angle + 0.02))

(defn -main []
  (js/createCanvas 710 400 js/WEBGL)
  (js/frameRate 30)
  (.draw js/p5 draw))
```

Quil

- Clojure(Script) wrapper around p5.js
- Embraces the Clojure dogma and functional programming.
- Gives a declarative syntax for working with primatives.

```clojure
(ns rotating-cube.core
  (:require [quil.core :as q :include-macros true]
            [quil.middleware :as m]))

(defn setup []
  (q/frame-rate 30)
  (q/color-mode :rgb)
  {:angle 0})

(defn update-state [state]
  (update state :angle (fn [angle] (+ angle 0.01))))

(defn draw-state [state]
  (q/background 0)
  (q/with-rotation [(:angle state) 1 1 0]
    (q/box 100)))    ;; Draw box of size 100



(defn -main []
  (q/defsketch main
    :host "rotating-cube"
    :size [710 400]
    :setup setup
    :renderer :p3d

    :update update-state
    :draw draw-state
    :middleware [m/fun-mode]))

(defn ^:export run-sketch []
  (-main))

(defn on-js-reload []
  )
```

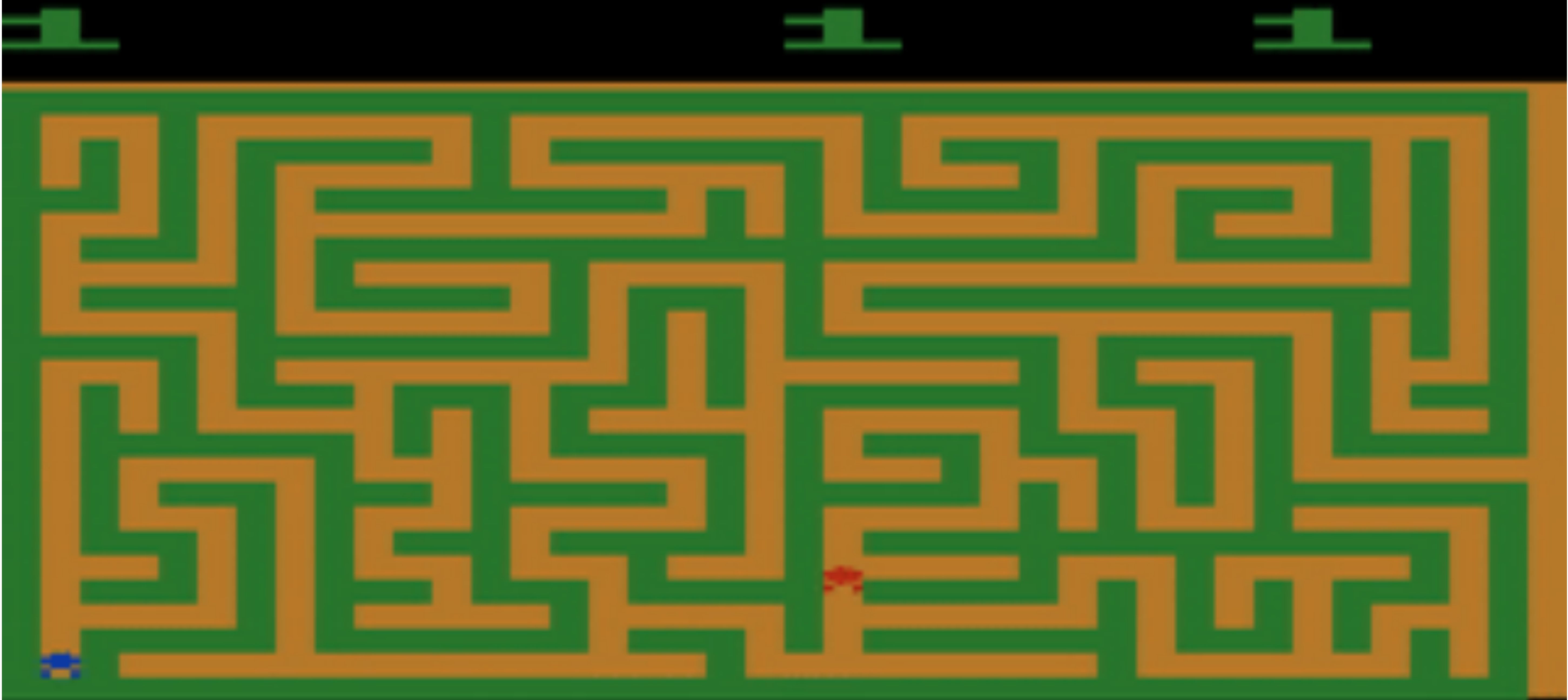
```clojure
(q/with-rotation [(:angle state) 1 1 0]
    (q/box 100))
```

Some history

- In 1980 Atari published Maze Craze
- In it you control a blob trying to beat another blob moving through the maze.
- Sometimes other blob might try and stop you.

Bit of an aside

> 2. GAME PLAY
>
> You're a cop confronting danger and suspense as you and your opponent wind your way across the city blocks. The first player to reach the exit on the right side of the maze wins the game.
>
> Throughout the game you may encounter armed robbers, blockades, and other obstacles to prohibit you from finishing your beat.


www.BANDICAM.com

What is the point of this?

- The maze is different upon every reboot of the console.
- It generates this maze randomly every time, leading to potentially thousands of different levels!

Level Generation

- It's becoming common to have procedurally generated levels in games.
- Games like Minecraft, The Binding of Isaac, and Shadows of Doubt utilize procedural generation so that they can have "unlimited" gameplay.

Basic Maze Generation

- Pick a cell to visit.
- Mark it visited.
- Find its neighbors
- For each unvisited neighbor (in random order), break down the adjoining walls.
- Recurse onto each unvisited neighbor.

Clojure Code

```
(let [
     init-grid (atom (->> (repeat mazeheight nil)
                          (mapv
                            (fn [_]
                              (into [] (repeat mazewidth {:visited false
                                                          :broken-walls #{}
                                                          }))))))
                                                          ]
```

```clojure
(defn mazegen2 [x y grid]
  (swap! grid assoc-in [y x :visited] true)
  (let [
        height (count @grid)
        width (count (first @grid))
        neighbors (shuffle [[:right (+ 1 x) y] [:left (- x 1) y] [:down x (+ 1 y)] [:up x (- y 1)]])
        ]

    (doseq [[dir nx ny] neighbors]
      (when (and (>= nx 0)
                 (>= ny 0)
                 (< nx width)
                 (< ny height))
        (let [c (get-in @grid [y x])
              n (get-in @grid [ny nx])

              visited (:visited n)
              bw (or (:broken-walls c) #{})
              nbw (or (:broken-walls n) #{})
              ]
          (when (not visited)
            (do
              (swap! grid assoc-in [y x :broken-walls] (conj bw dir))
              (swap! grid assoc-in [ny nx :broken-walls] (conj nbw (get opmap dir)))

              (mazegen2 nx ny grid))))))))
```

Drawing the Maze

- First we need something to draw boxes.

```clojure
(defn big-box [box-width inner-box-width broken-walls]
  (q/with-translation [0 0 0]
    (q/box inner-box-width)
    )
  (q/with-translation [(* (- box-width 1) inner-box-width) 0 0]
    (q/box inner-box-width)
    )
  (q/with-translation [0 (* (- box-width 1) inner-box-width) 0]
    (q/box inner-box-width)
    )
  (q/with-translation [(* (- box-width 1) inner-box-width) (* (- box-width 1) inner-box-width) 0]
    (q/box inner-box-width)
    )

  (if (not (contains? broken-walls :up))
      (q/with-translation [(* 0.5 (- box-width 1) inner-box-width) 0 0]
        (q/box (* (- box-width 2) inner-box-width ) inner-box-width inner-box-width)
       )
     ; )
    )
  (if (not (contains? broken-walls :left))
      (q/with-translation [0 (* 0.5 (- box-width 1) inner-box-width) 0]
        (q/box inner-box-width (* inner-box-width (- box-width 2)) inner-box-width)
        )
      ;)
    )
  (if (not (contains? broken-walls :right))
      (q/with-translation [(* inner-box-width (- box-width 1)) (* 0.5 (- box-width 1) inner-box-width) 0]
        (q/box inner-box-width (* inner-box-width (- box-width 2)) inner-box-width)
        )
    )
  (if (not (contains? broken-walls :down))
    (q/with-translation [(* 0.5 (- box-width 1) inner-box-width)  (* (- box-width 1) inner-box-width) 0]
      (q/box (* (- box-width 2) inner-box-width) inner-box-width inner-box-width))))
```

Drawing the maze

- Next, we need to actually draw the maze part!

```
(defn draw-maze [maze block-width wall-length]
  (let [maze-height (count maze)
        maze-width (count (first maze))
        ]
    (doseq [y (range maze-height)]
      (doseq [x (range maze-width)]
        (q/with-translation [(* x block-width wall-length) (* y block-width wall-length) 0]
          (big-box t wall-length block-width (get-in maze [y x :broken-walls]))))))))
```

What about 2D?

- Wouldn't it be neat if we could fundamentally have the same core logic, but alternate between 2D and 3D on the fly?
- Multimethods to the rescue!

Defining a multimethod

```
(defmulti big-box (fn [t _ _ _]
                    t))
```

Create a 2D version

```
(defmethod big-box :2d [t box-width inner-box-width broken-walls]
  (q/rect 0 0 inner-box-width inner-box-width)
  (q/rect (* (- box-width 1) inner-box-width) 0 inner-box-width inner-box-width)
  (q/rect 0 (* (- box-width 1) inner-box-width) inner-box-width inner-box-width)
  (q/rect (* (- box-width 1) inner-box-width) (* (- box-width 1) inner-box-width) inner-box-width inner-box-width)

  (if (not (contains? broken-walls :up))
      (q/rect (* 0.5 (- box-width 3) inner-box-width) 0 (* (- box-width 2) inner-box-width )  inner-box-width))

  (if (not (contains? broken-walls :down))
      (q/rect (* 0.5 (- box-width 3) inner-box-width) (* (- box-width 1) inner-box-width) (* (- box-width 2) inner-box-width )  inner-box-width)
    )
  (if (not (contains? broken-walls :left))
      (q/rect 0 (* 0.5 (- box-width 3) inner-box-width) inner-box-width (* inner-box-width (- box-width 2)) ))
  (if (not (contains? broken-walls :right))
      (q/rect (* inner-box-width (- box-width 1)) (* 0.5 (- box-width 3) inner-box-width) inner-box-width (* inner-box-width (- box-width 2)) )))
```

Add an argument to the 3D version

```
(defmethod big-box :3d [t box-width inner-box-width broken-walls]
```

Pass the parameter based on state

```
(defn draw-maze [t maze block-width wall-length]

...

(defn draw
  (let [
        ...
         mode (:mode state)
         ]
           (draw-maze mode maze block-width num-blocks))
```

Aside about multimethods

- While popular in Clojure, can be used in basically any language.

```python
def value_dispatch(dispatch_key_func, dispatch_arg_index=0):
    def value_dispatch_decorator(func):
        func._value_dispatch = {}
        def wrapper(*args, **kwargs):
            key = dispatch_key_func(args[dispatch_arg_index])
            if key in func._value_dispatch:
                return func._value_dispatch[key](*args, **kwargs)
            else:
                return func(*args, **kwargs)
        def register(key):
            def inner(f):
                func._value_dispatch[key] = f
                return f
            return inner
        wrapper.register = register
        return wrapper
    return value_dispatch_decorator
```

```python
@value_dispatch(lambda x: x['type'], dispatch_arg_index=2)
def my_function(first_arg, second_arg, data: dict):
    raise ValueError(f"Invalid command: {data['type']}")

@my_function.register("profile")
def _(first_arg, second_arg, data):
    print("howdy")
```

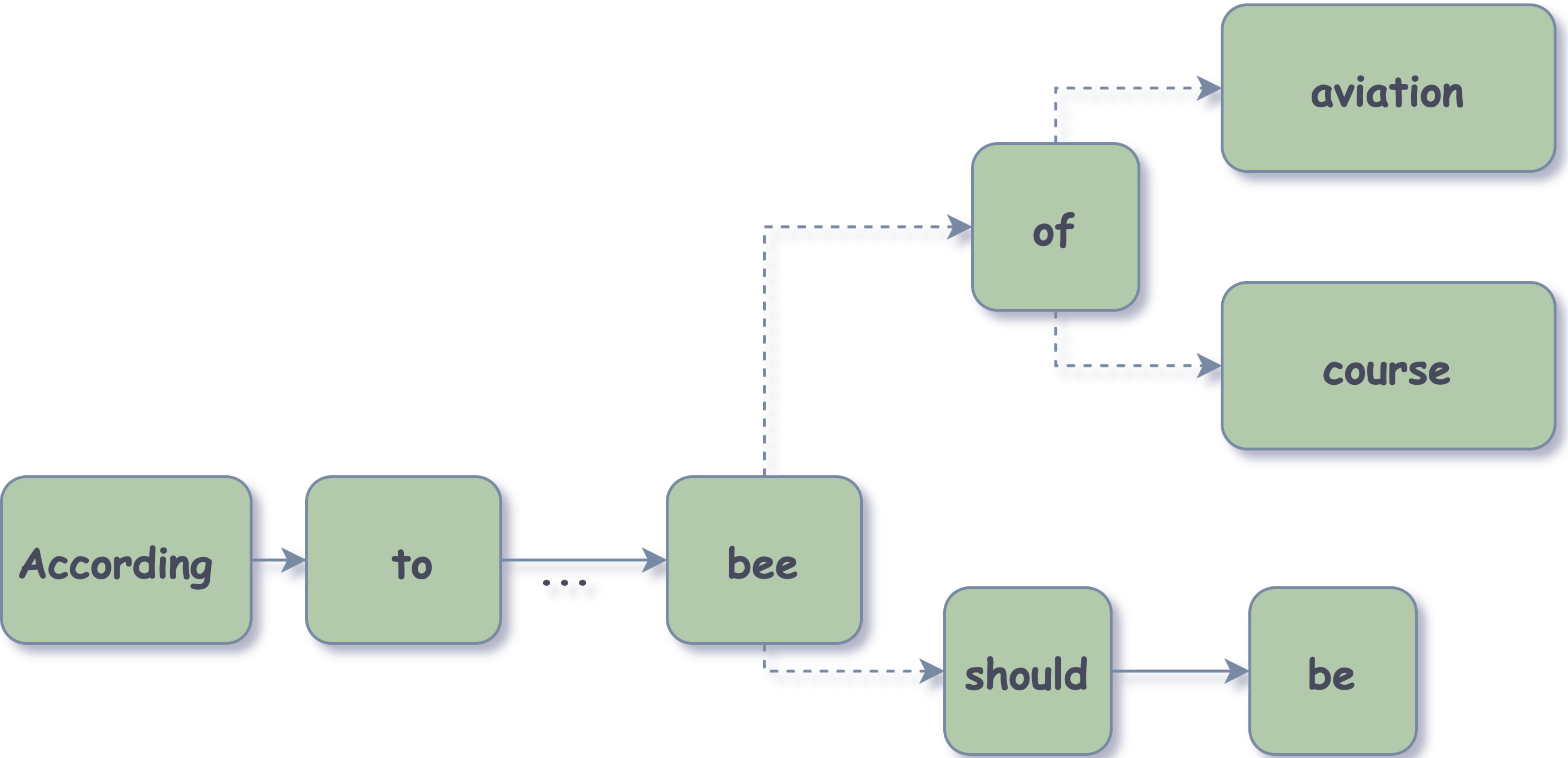
More generation

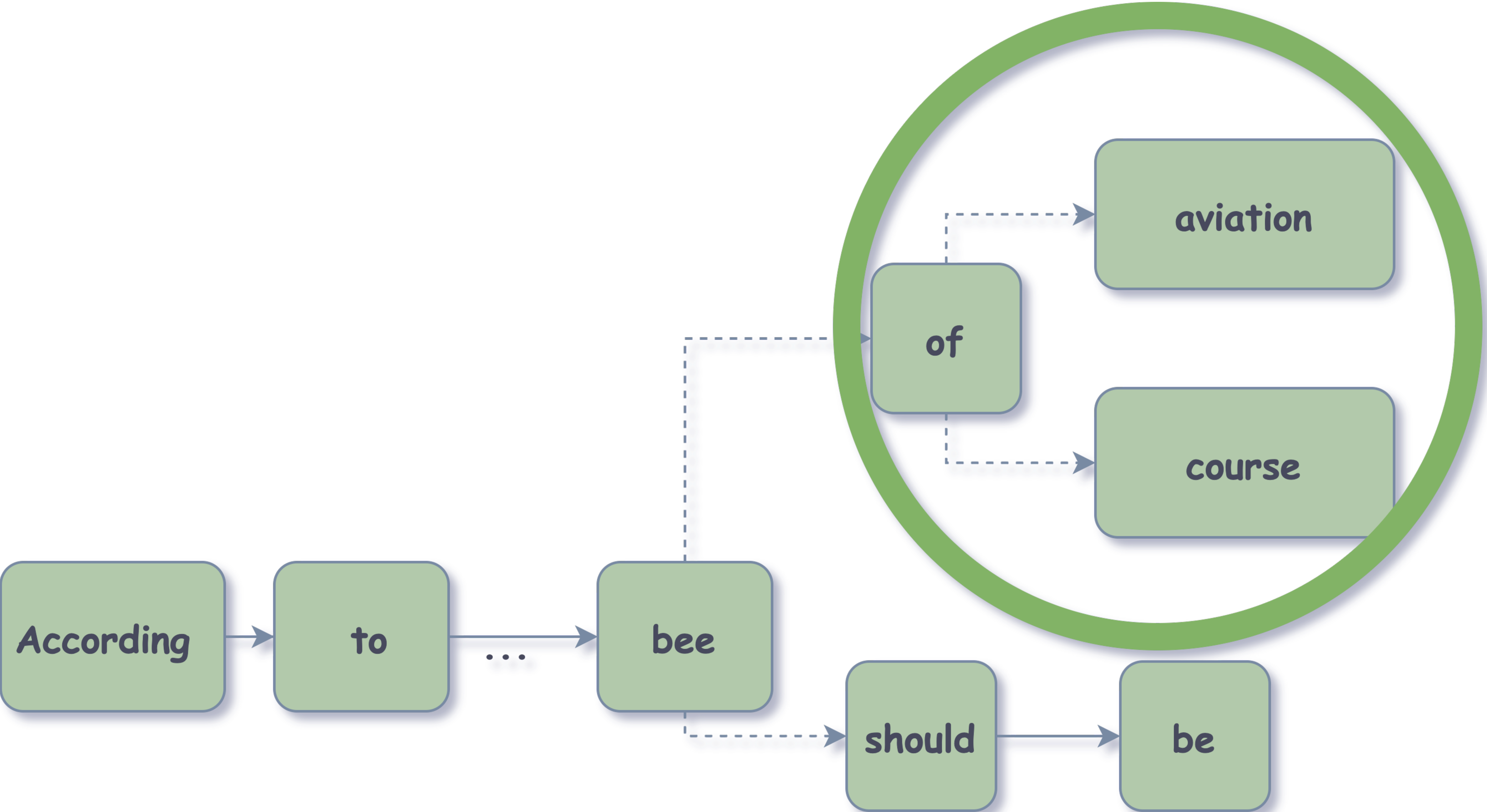
- Nearly anything involving patterns and randomness can be a level generator.
- What works for you is going to depend on what type of thing you're making.

Markov Chains

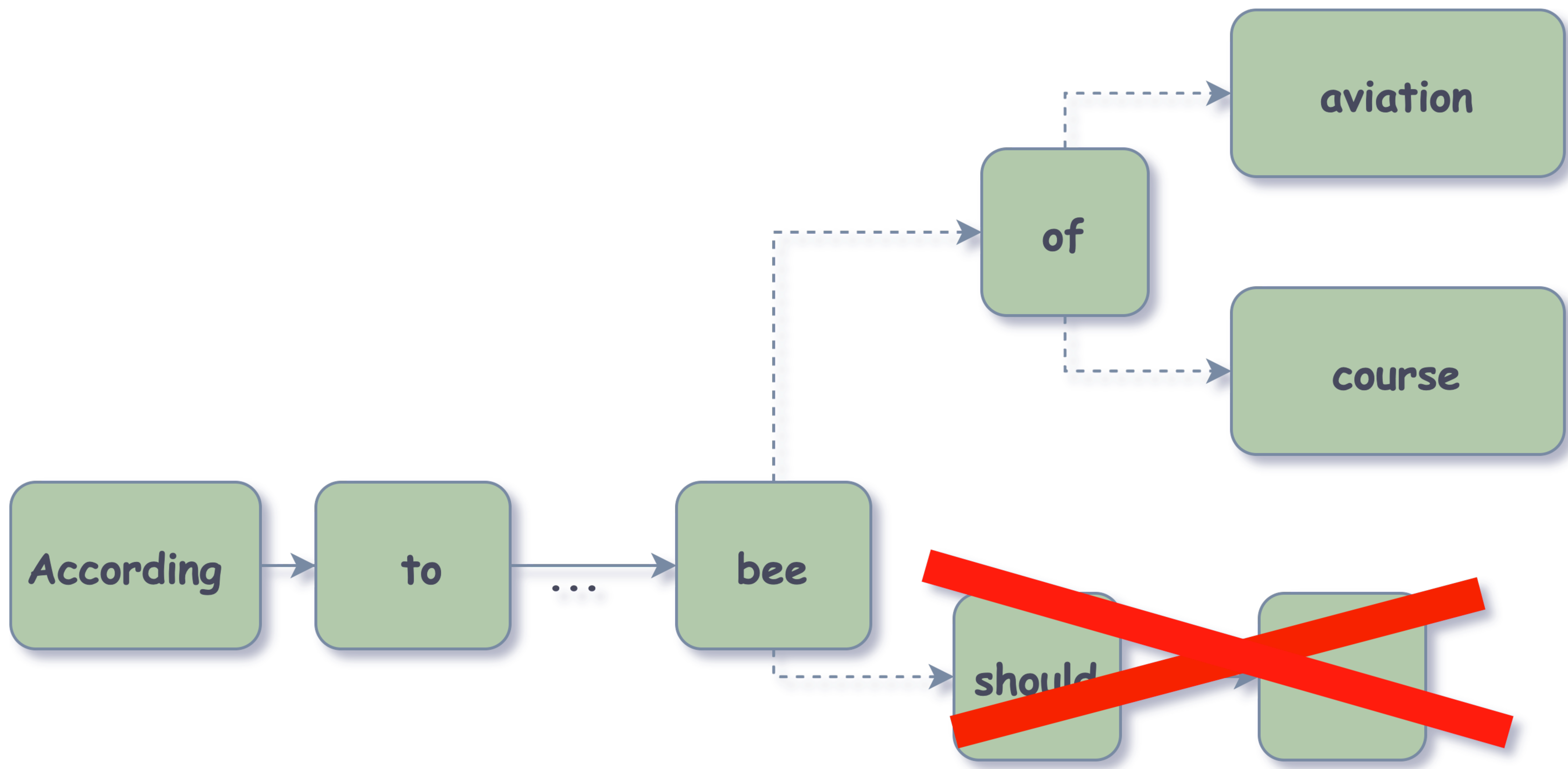
- There's a *lot* to Markov chains that I won't get into.
- Let's make a bad text generator.

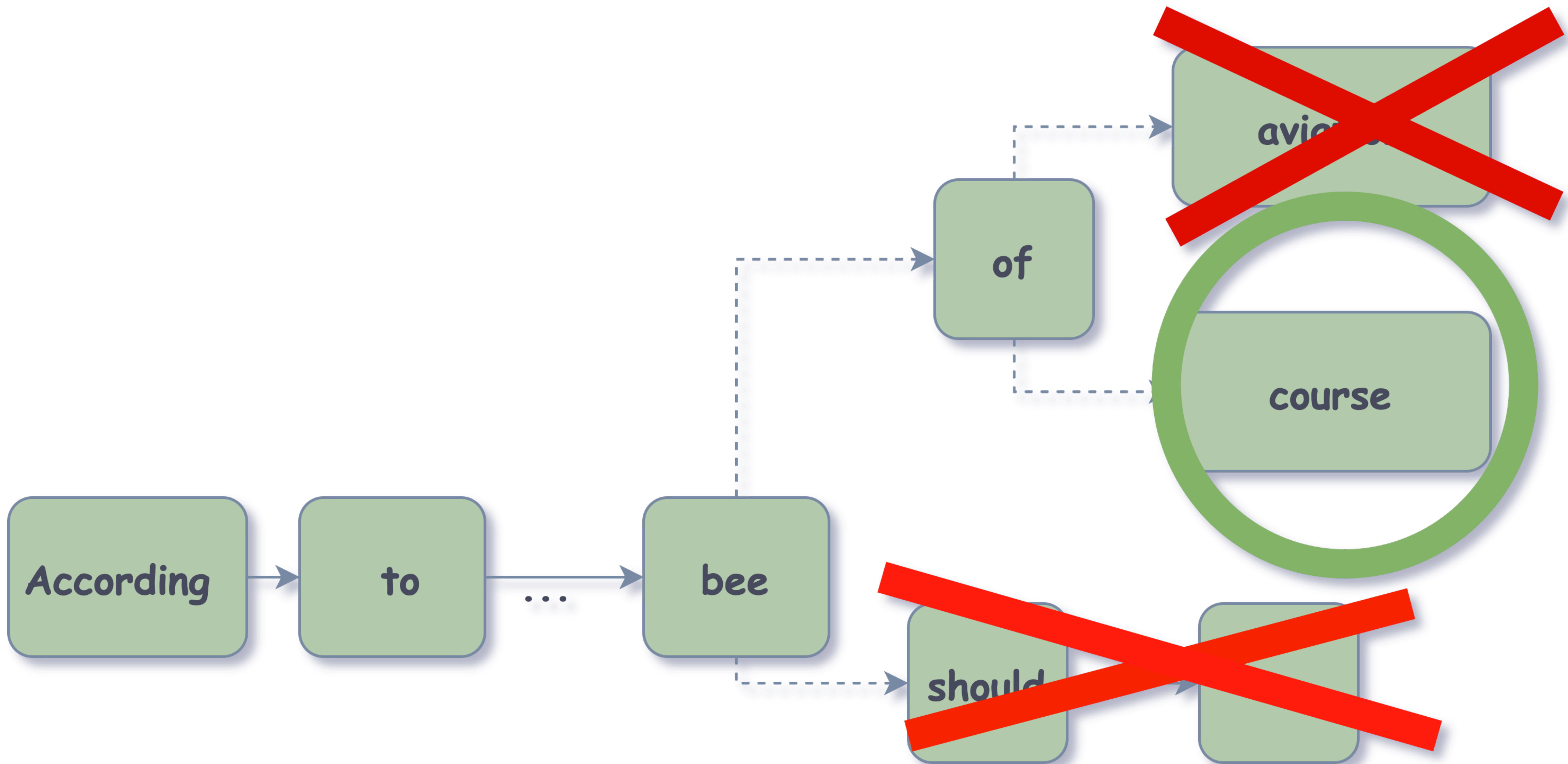
```clojure
(defn generate-markov-chain [words]
  (let [freq-map (reduce (fn [m [k v]]
                           (update m k conj v))
                         {}
                         (partition 2 1 words))
        start-words (filter #(= (count %) 1) words)]
    (fn []
      (loop [current-word (rand-nth start-words)
             result []]
        (if-let [next-words (get freq-map current-word)]
          (let [next-word (rand-nth next-words)]
            (recur next-word (conj result next-word)))
          result)))))

(defn generate-text [filename]
  (let [text (-> filename slurp (str/split #" "))
        chain (generate-markov-chain text)]
    (apply str (interpose " " (chain)))))
```


According
to
...
bee
of
aviation
course
should
be



aviation
of
course
According
to
...
bee
should
be


aviation
of
course
According
to
...
bee
should



According
to
...
bee
of
course
should

Limitations to Markov Chains

- Markov Chains are cool, but they're sort of limiting.
- Output is usually one-dimensional, difficult to adapt to much other than text generation.

Wave Function Collapse (WFC)

- Sort of an "N-Dimensional Markov Chain".
- Can be used to generate all manners of things.
- Unjustifiably cool name.

WFC Basic Algorithm.

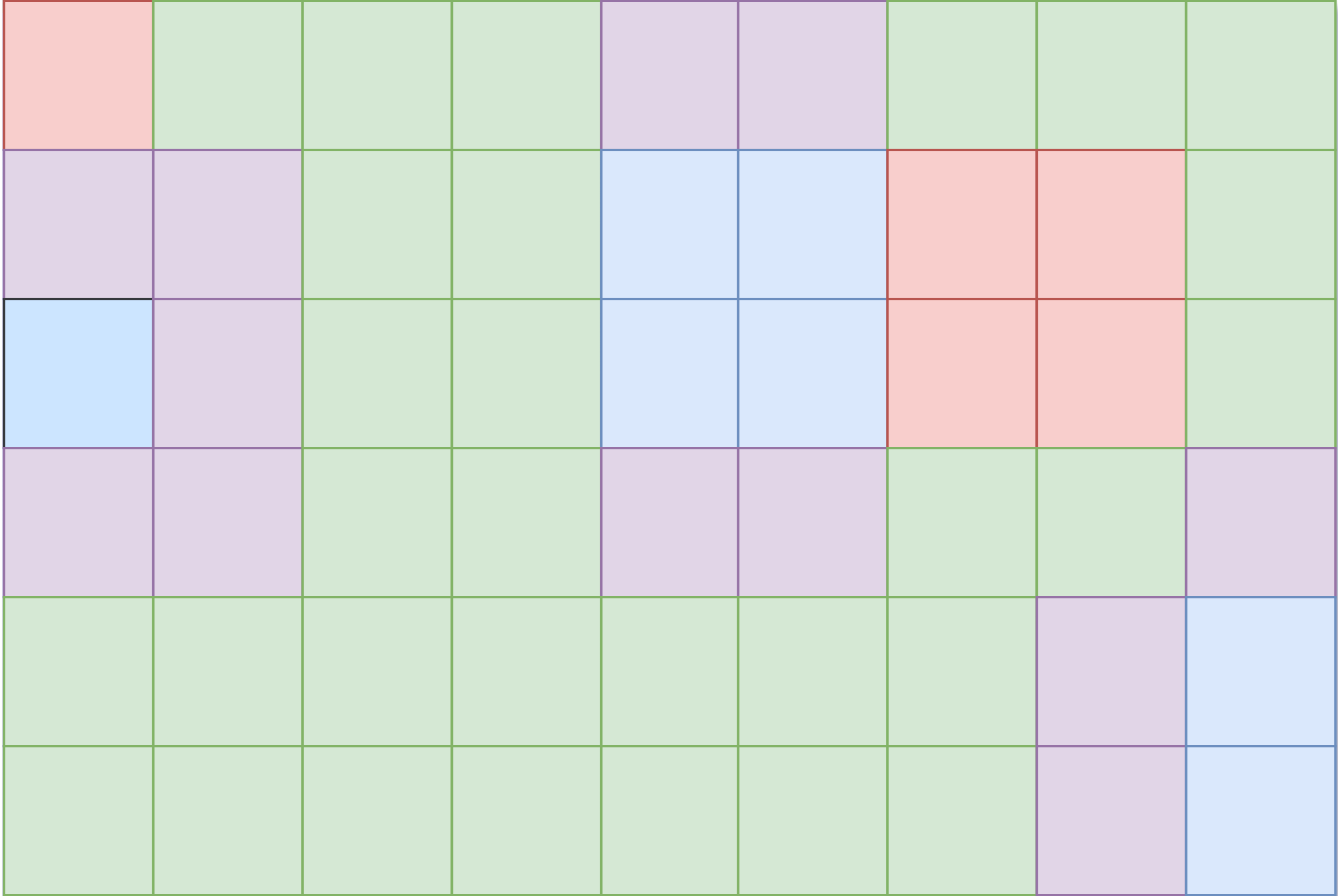
- Assume some sort of adjacency rules.
- In a grid, all possible arrangements of those rules exist simultaneously in superposition.
- "Collapse" one of the waves by picking one of the possible positions.
- Repeat until you have a grid that you're happy with.

10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10

	5	7	10	10	10	10	10	10
5	5	7	10	10	10	10	10	10
7	7	7	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10

	2	5	10	10	10	10	10	10
	2	6	10	10	10	10	10	10
4	4	4	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10

			5	4	10	10	10	10
			5	4	10	10	10	10
			3	4	10	10	10	10
2	2	3	2	5	10	10	10	10
6	6	7	8	7	10	10	10	10
10	10	10	10	10	10	10	10	10



WFC Adjacency

- Typically rules are defined by some sort of sample input.
- Similar to a Markov chain.

Example

- Courtesy of `allison-casey` and `mxgmn` on Github.
  - https://github.com/mxgmn/WaveFunctionCollapse
  - https://github.com/allison-casey/wavefunctioncollapse-clj

Further Reading

- Perlin Noise
  - Minecraft uses modified versions of this.
  - Effectively a "gradient noise" generation algorithm.
- More about WFC
  - https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/

Contact Me

- thomas@gebert.app