Браузерный игровой движок как pet-проект

Меня зовут Михаил Реммеле

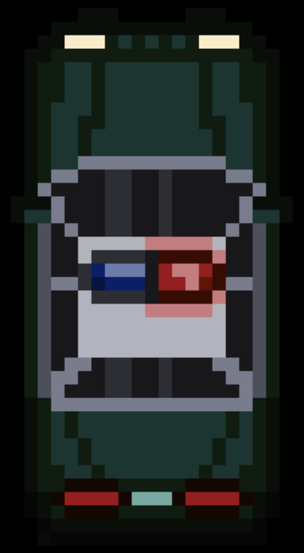
Работаю frontend разработчиком

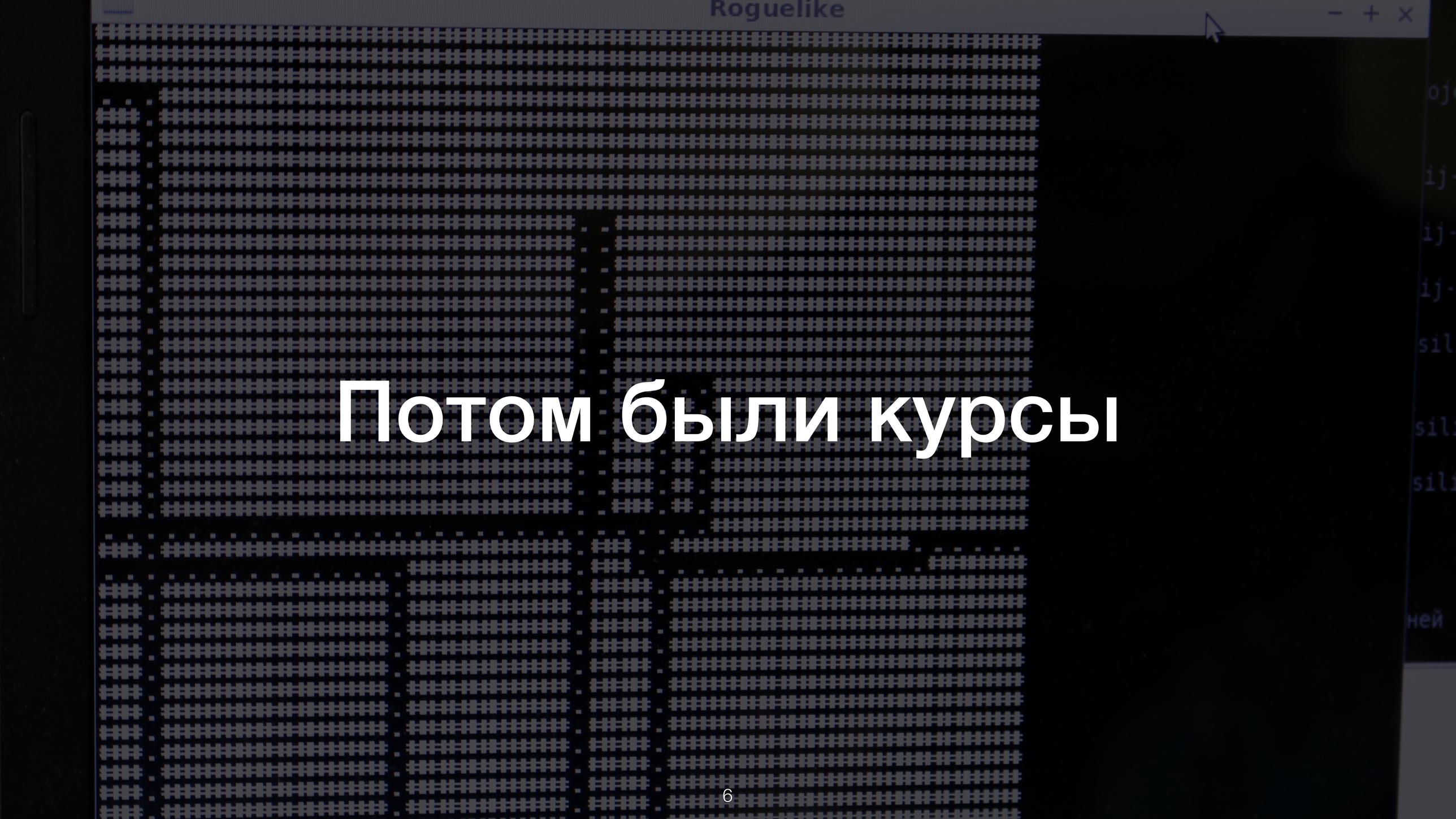
В Сбере

Почему игры

Все началось тут:

Но игры я не писал





Потом были курсы

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * **Repository name** *

michailRemmele / game-engine

Great repository names are short and memorable. Need inspiration? How about jubilant-octo-telegram?

Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Ну всё. Я тоже хочу

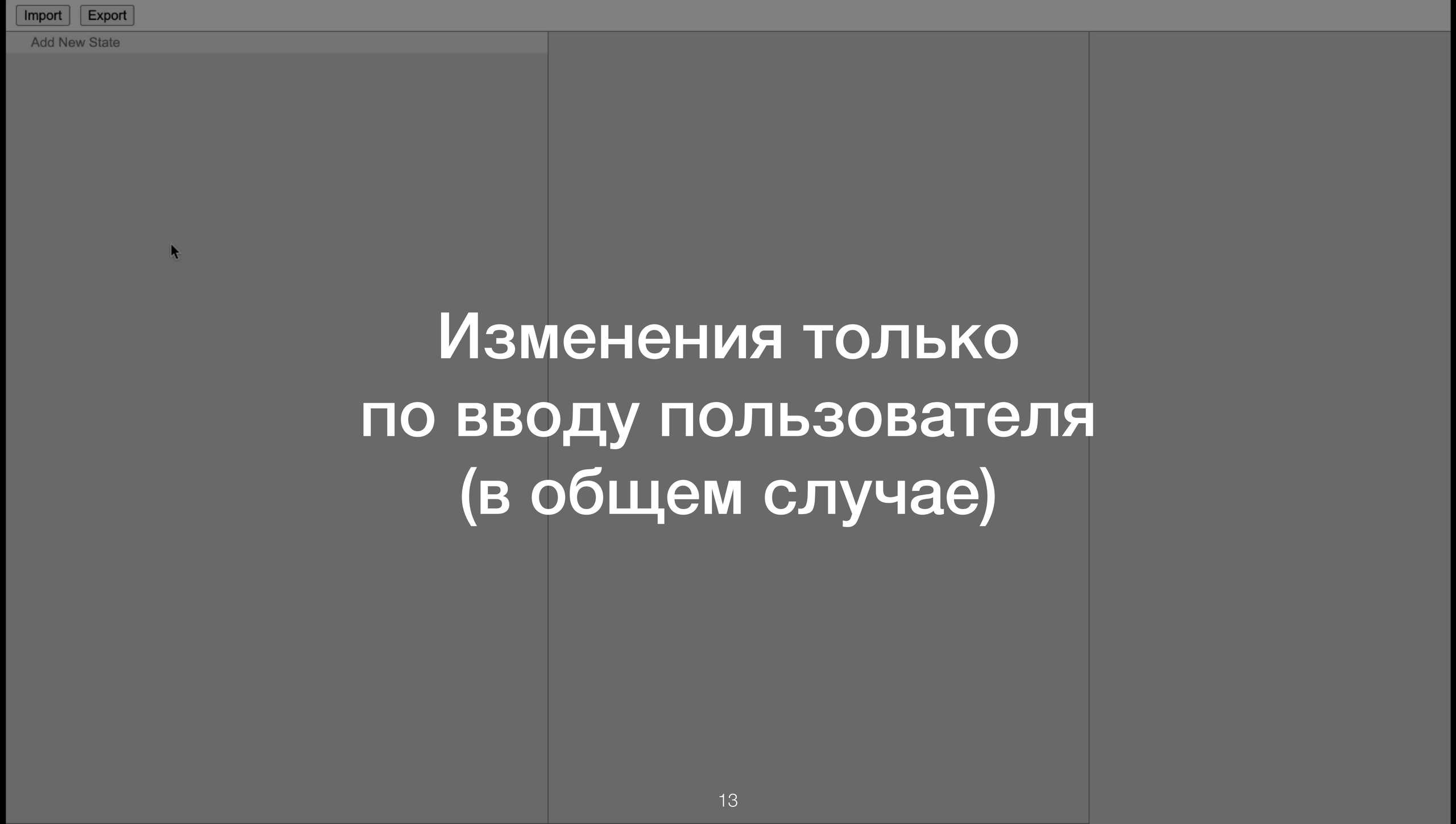
При чем тут Web и JS

- Можно использовать полученные знания на работе
- Легко делиться результатами

А какая цель

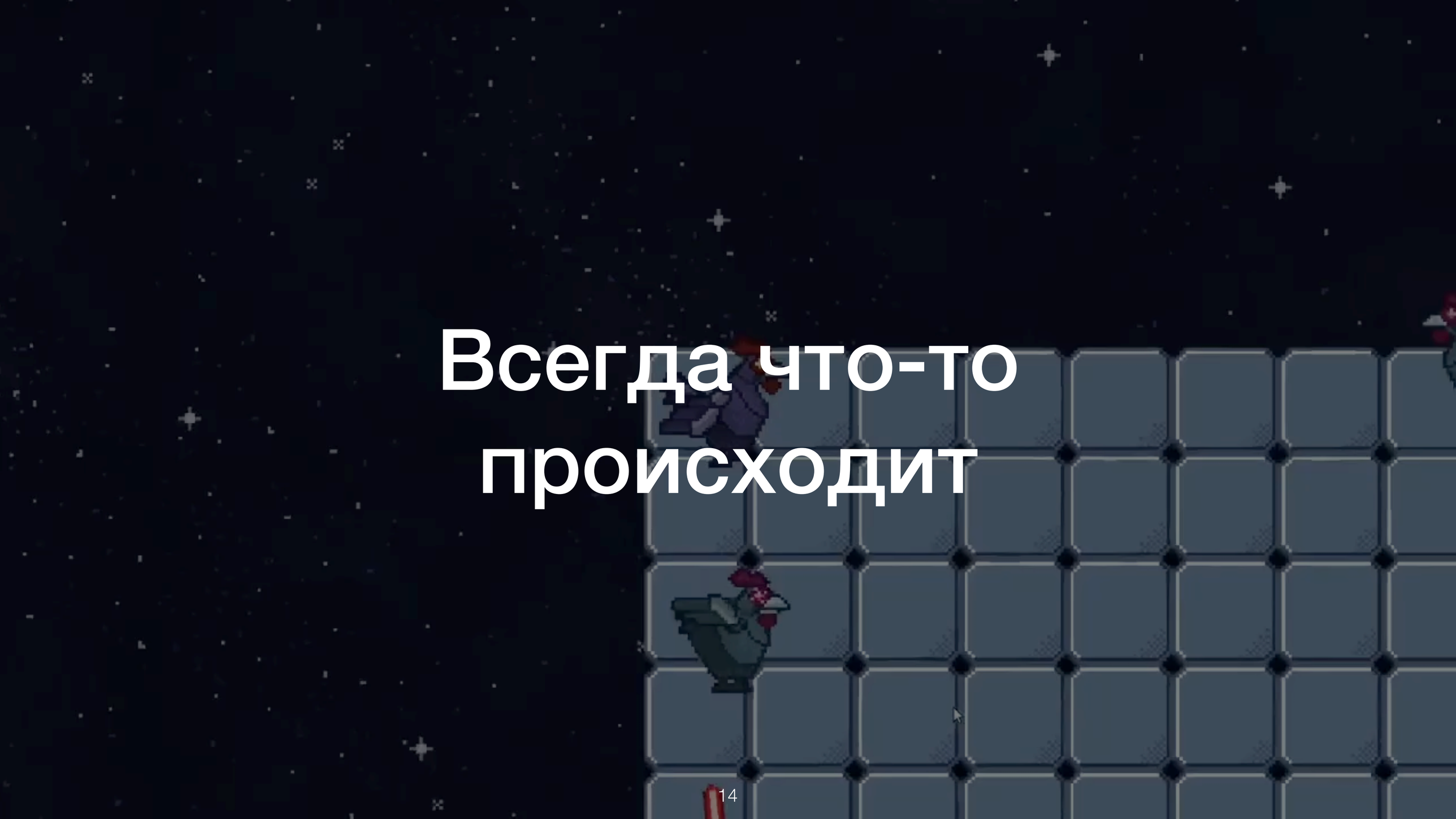
- Разобраться как устроена разработка игр с нуля
- Создать MVP игрового движка

Чем игра отличается от обычного приложения

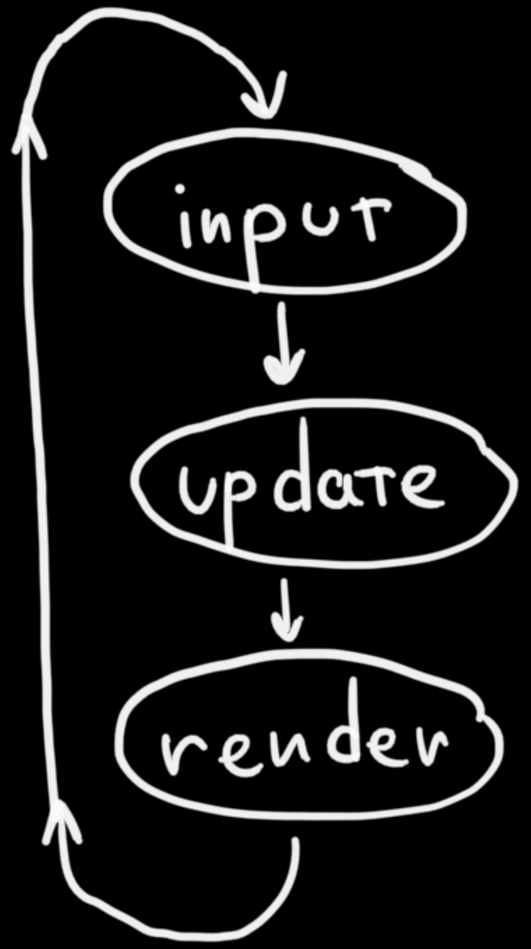


Изменения только по вводу пользователя (в общем случае)

Всегда что-то происходит



Игры работают в цикле



```
let previous = performance.now();

requestAnimationFrame(function tick(current) {
  const deltaTime = current - previous;
  previous = current;

  input();
  update(deltaTime);
  render();

  requestAnimationFrame(tick);
});
```

Игры работают в цикле

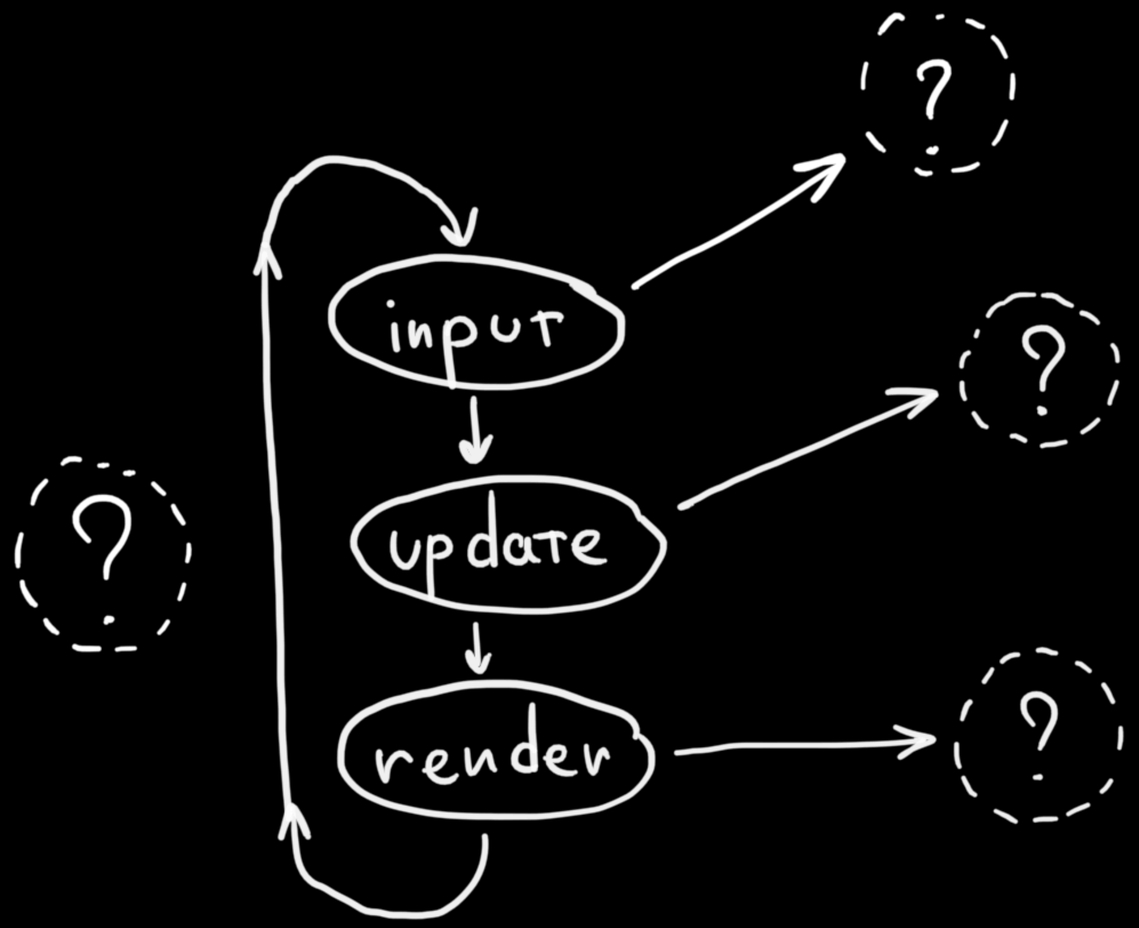
Что такое игровой движок

Набор инструментов

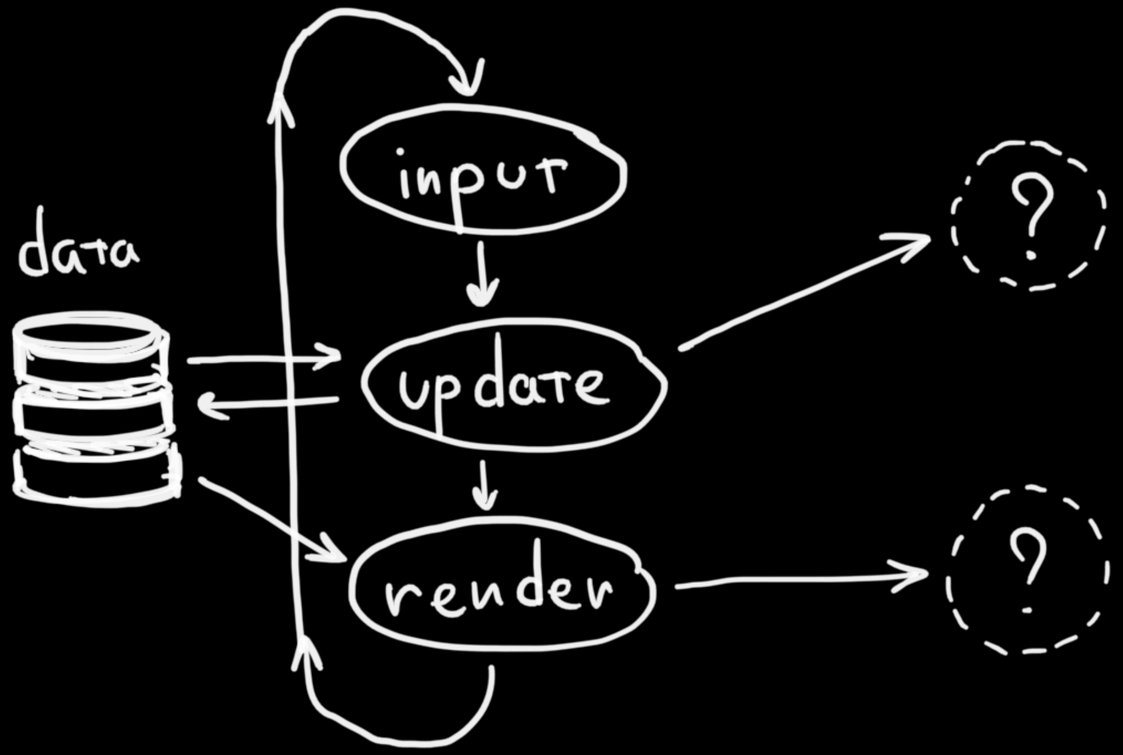
Всегда ли нужен движок, чтобы сделать игру


НЕТ
ARROWS KEYS-MOVE
ENTER-FIRE
FIRES : 2

Приступим

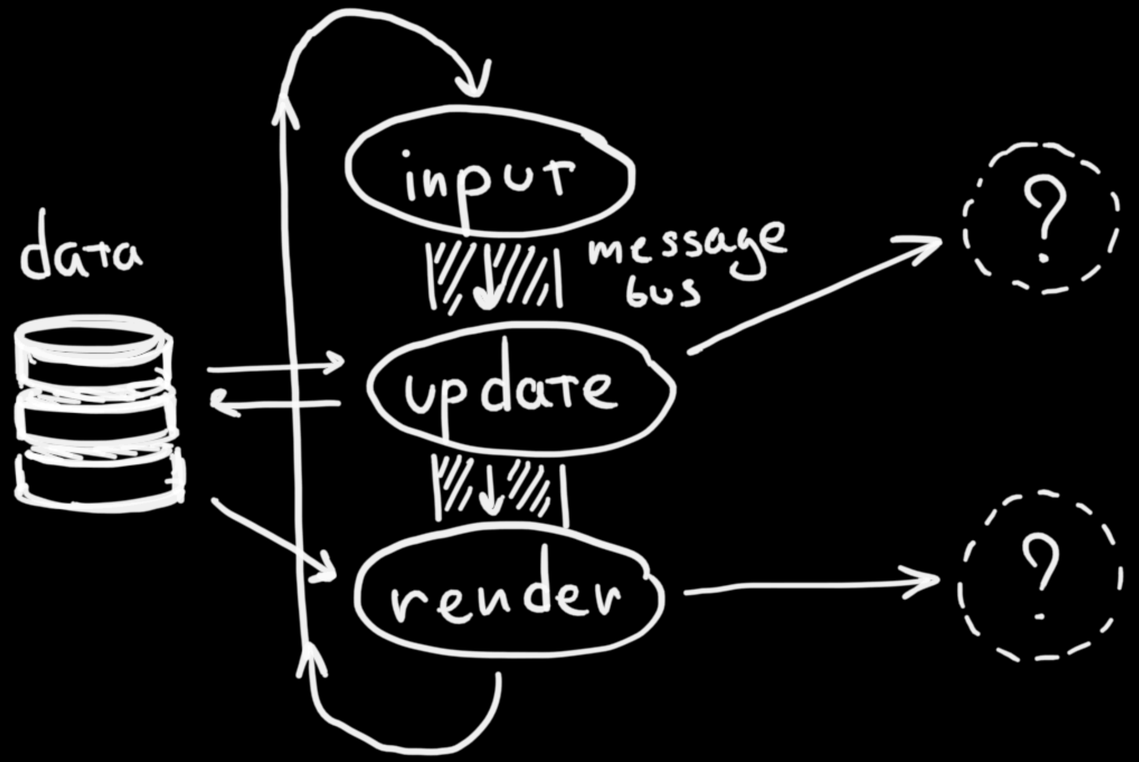


Начнем с
декомпозиции

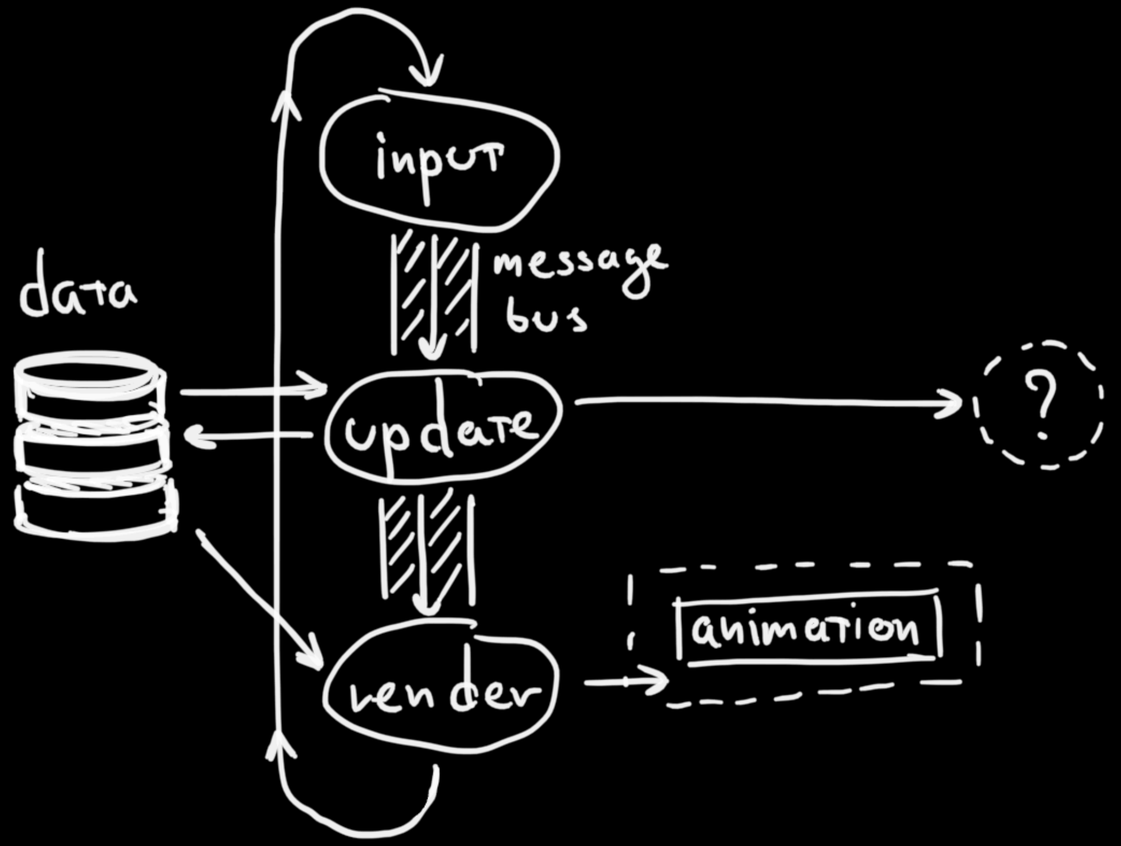


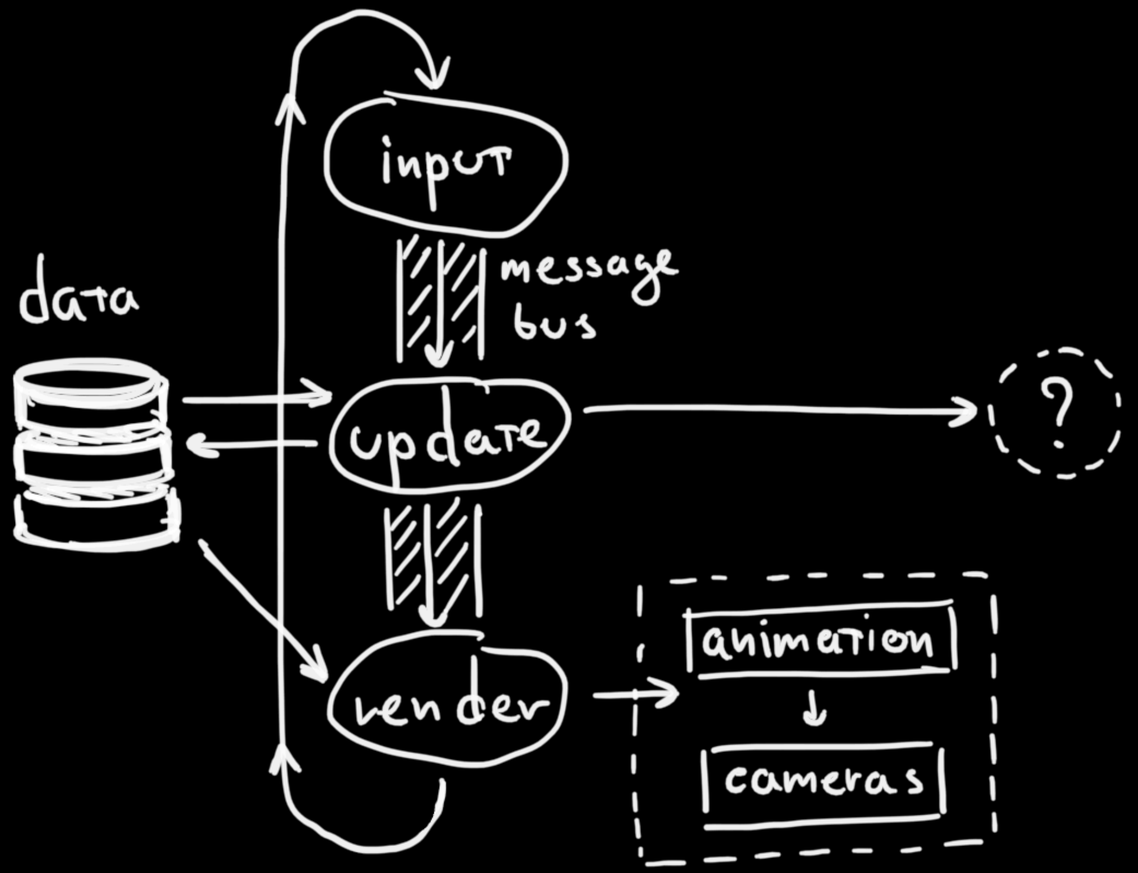
Модель данных

Средство для общения подсистем



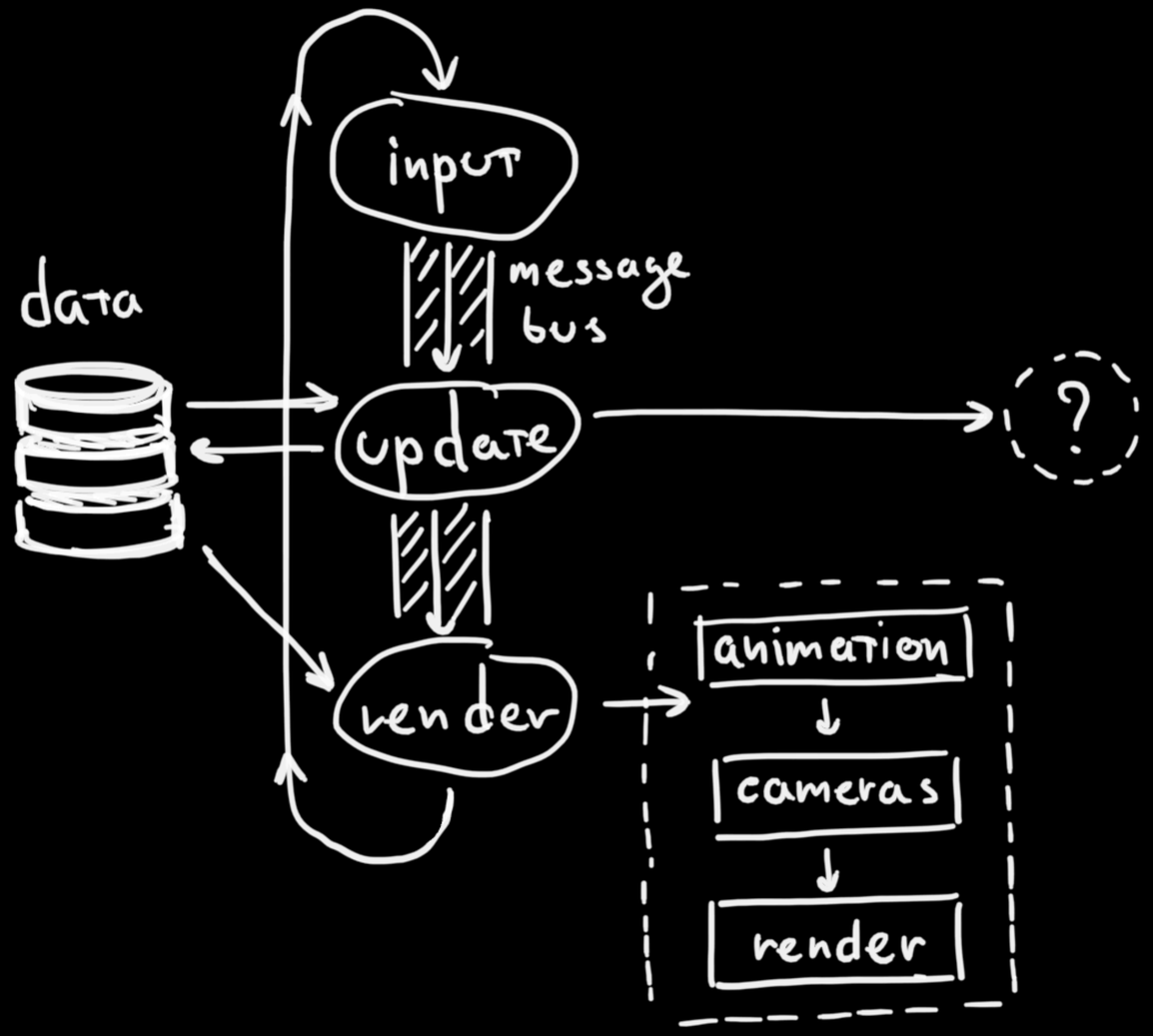
Анимация

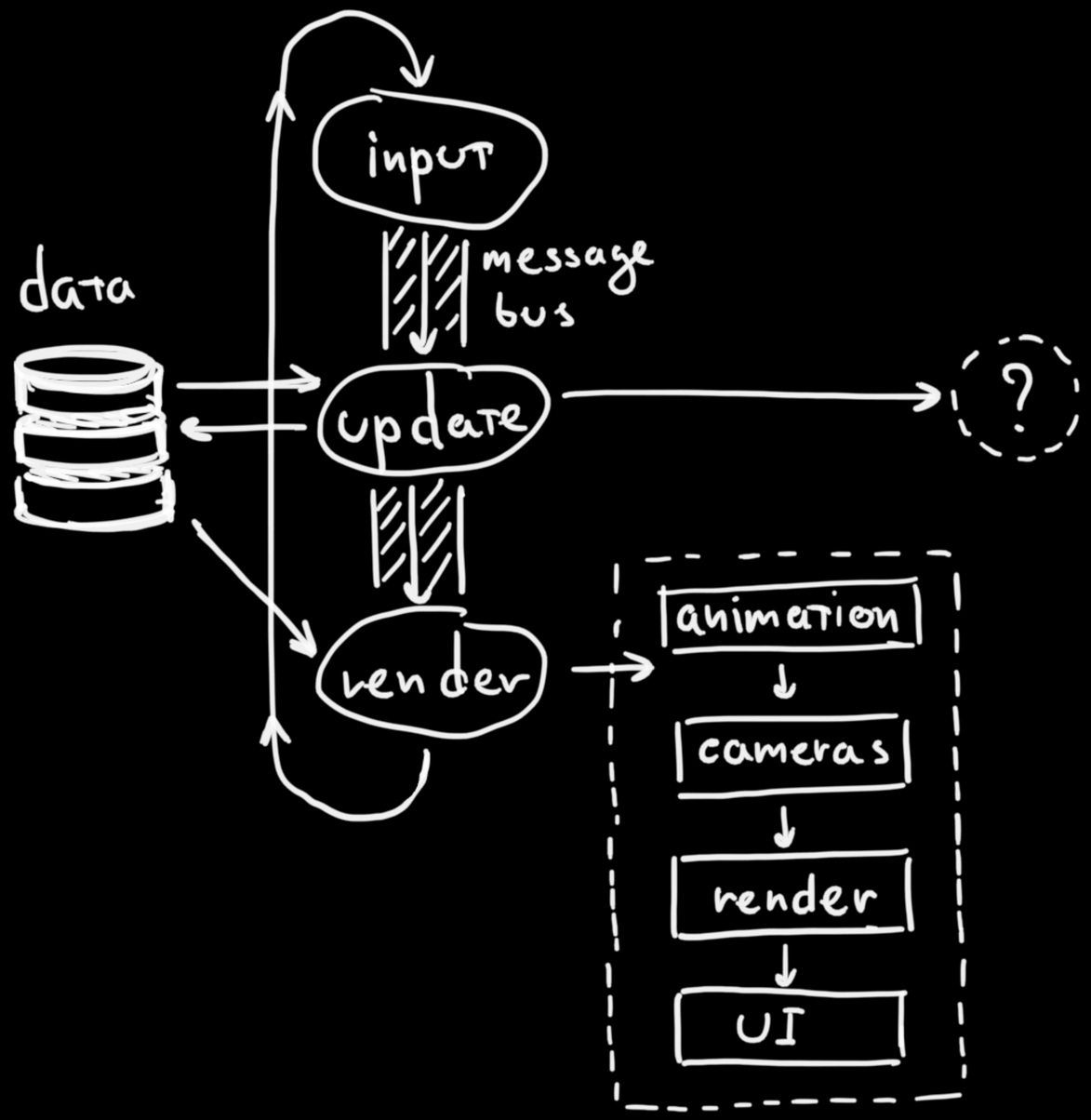




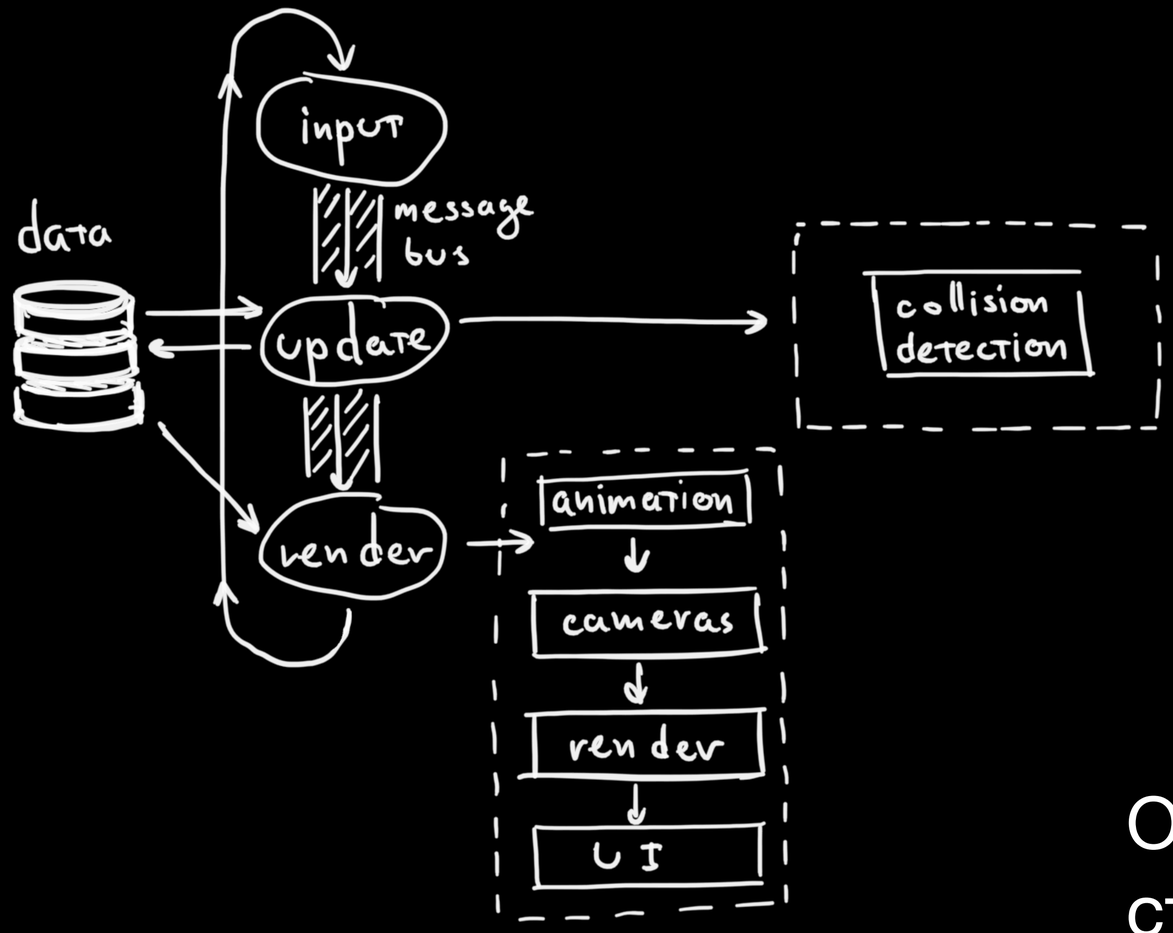
Камеры

Отрисовка


input
message bus
data
update
?
render
animation
cameras
render

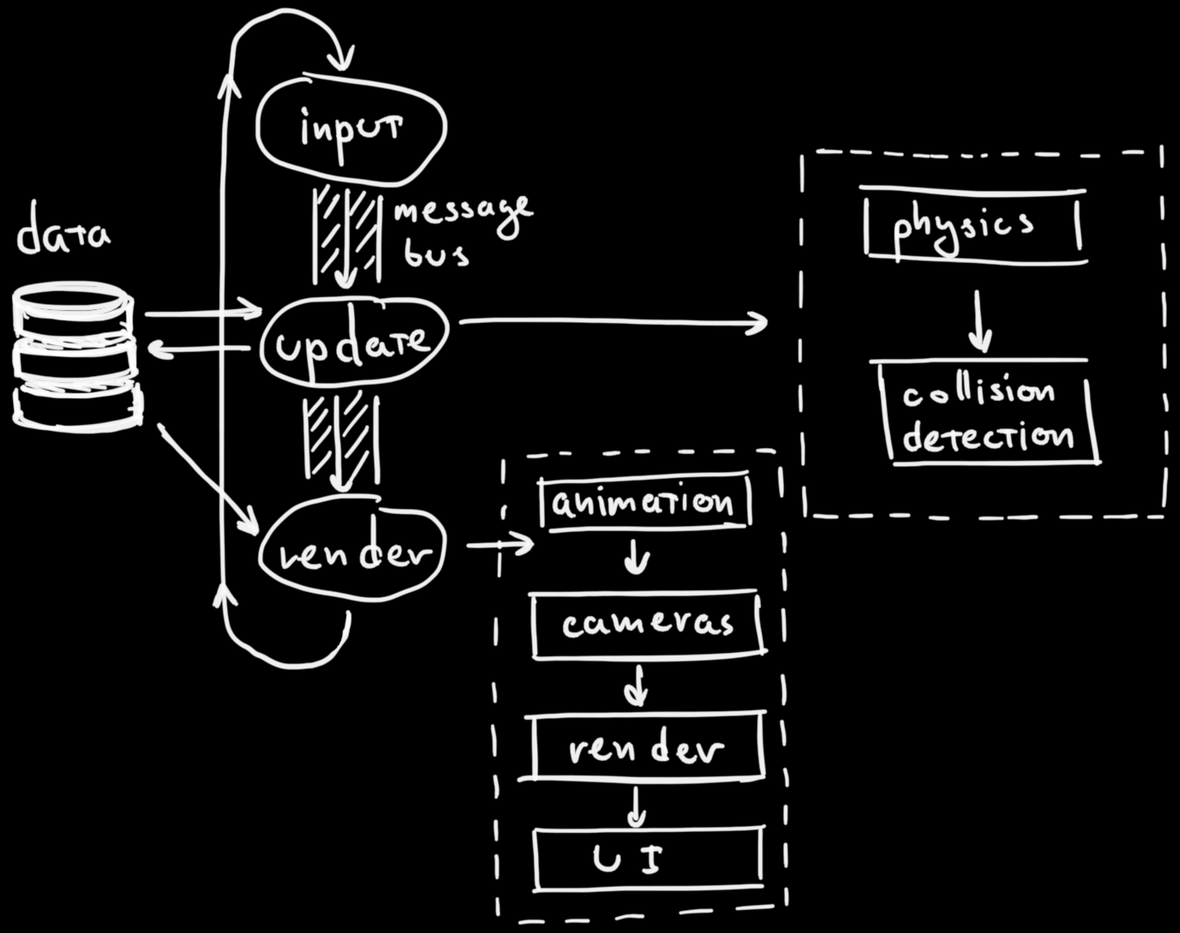

input
message
bus
data
update
?
render
animation
cameras
render
UI

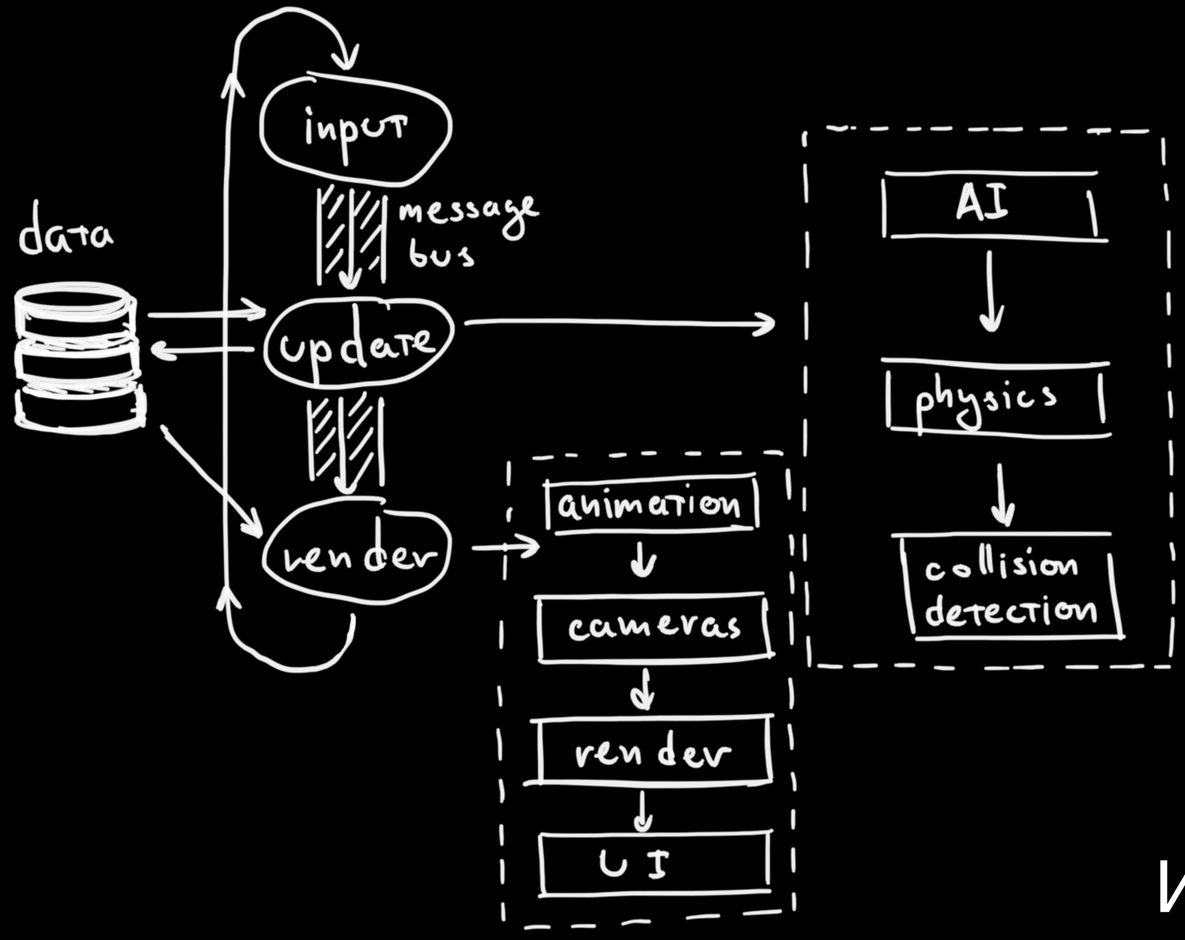
Интерфейс



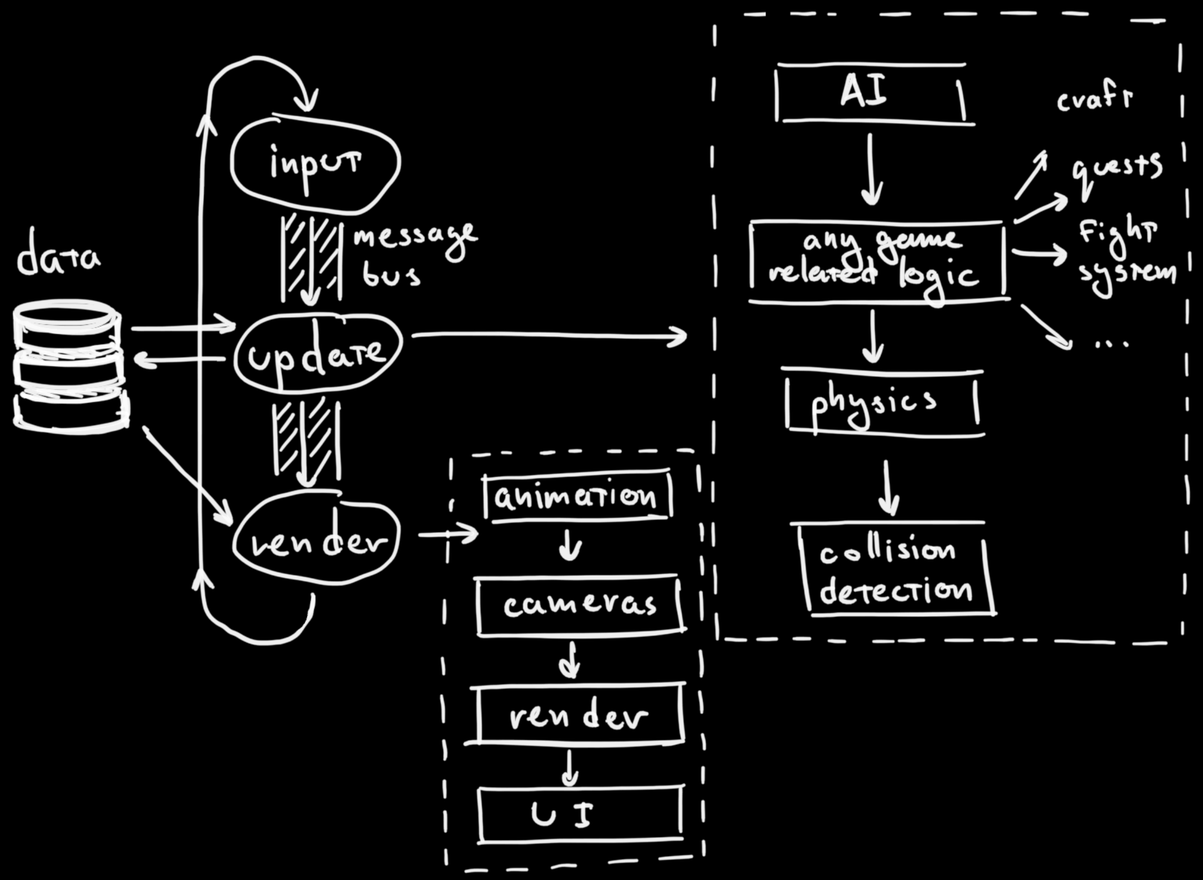
Обнаружение столкновений

Физика

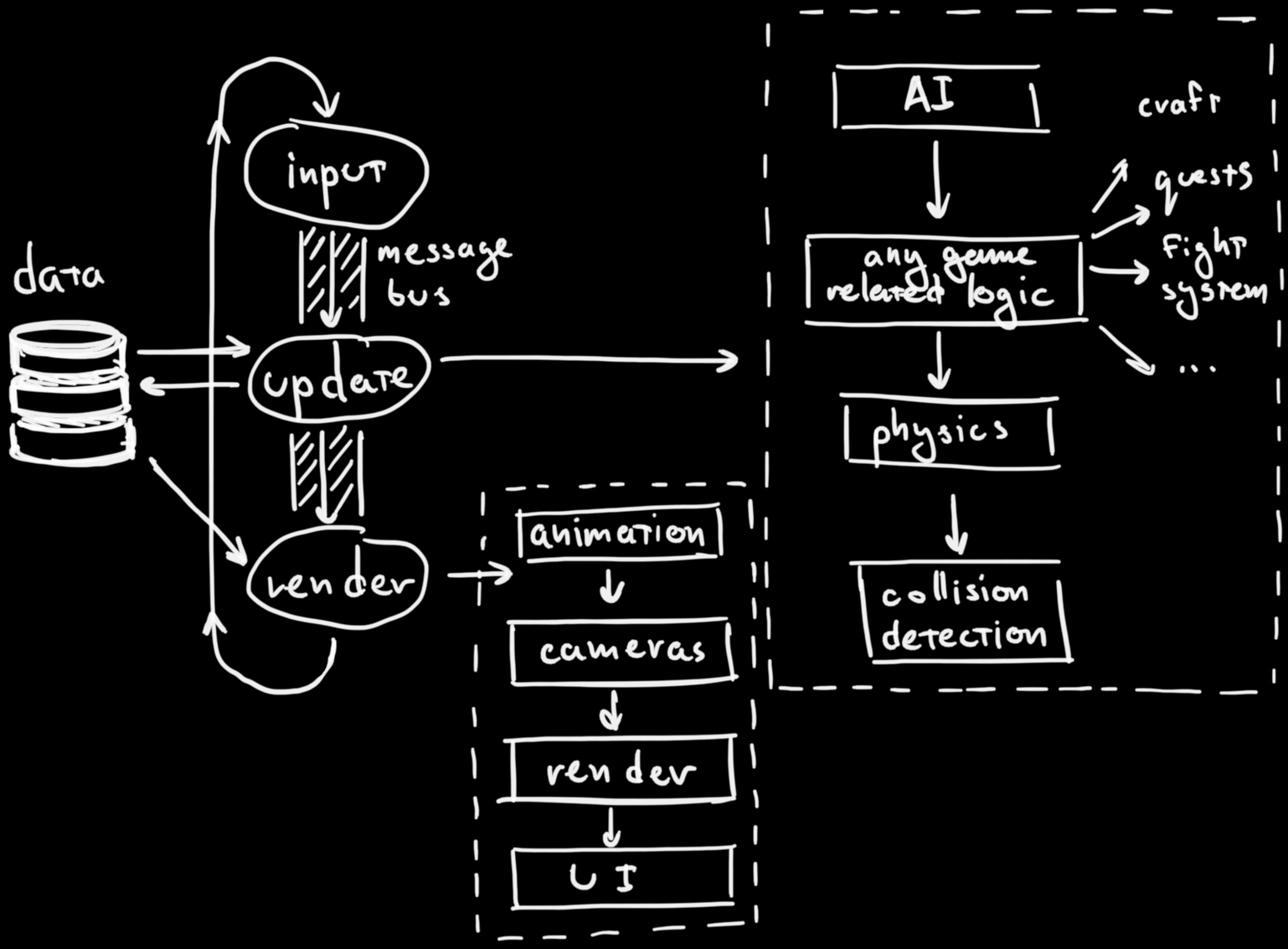

input
message bus
data
update
render
physics
collision detection
animation
cameras
render
UI



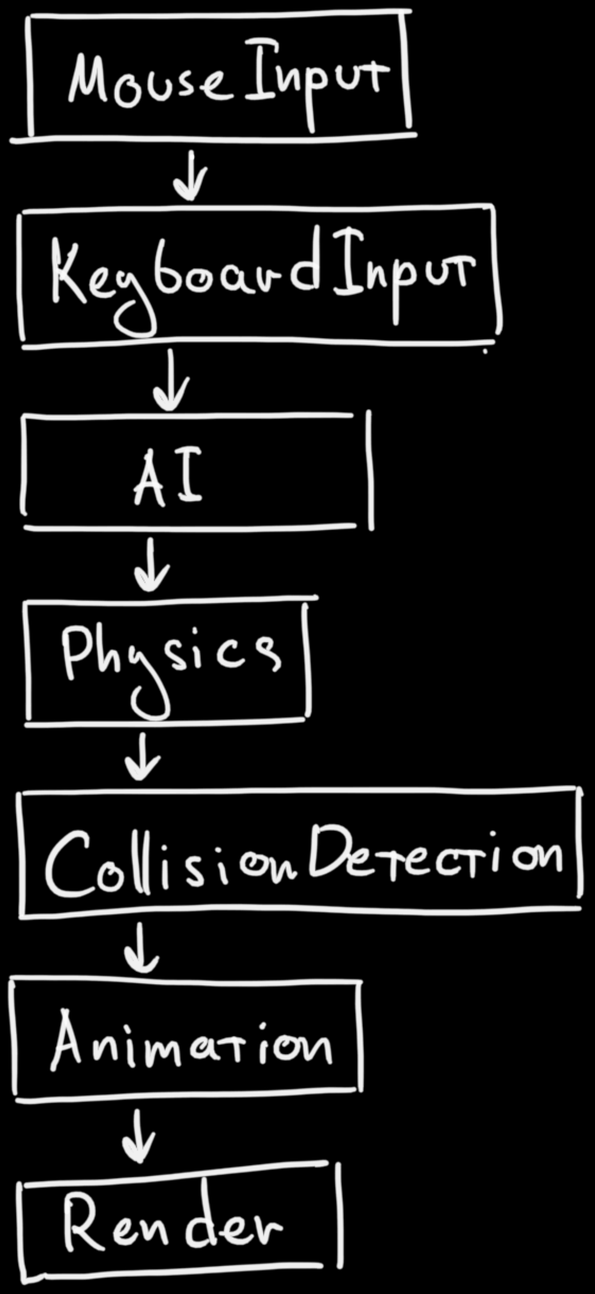
Искусственный интеллект


input
message bus
data
update
render
AI
craft
quests
any game related logic
fight system
...
physics
collision detection
animation
cameras
render
UI

И так далее…

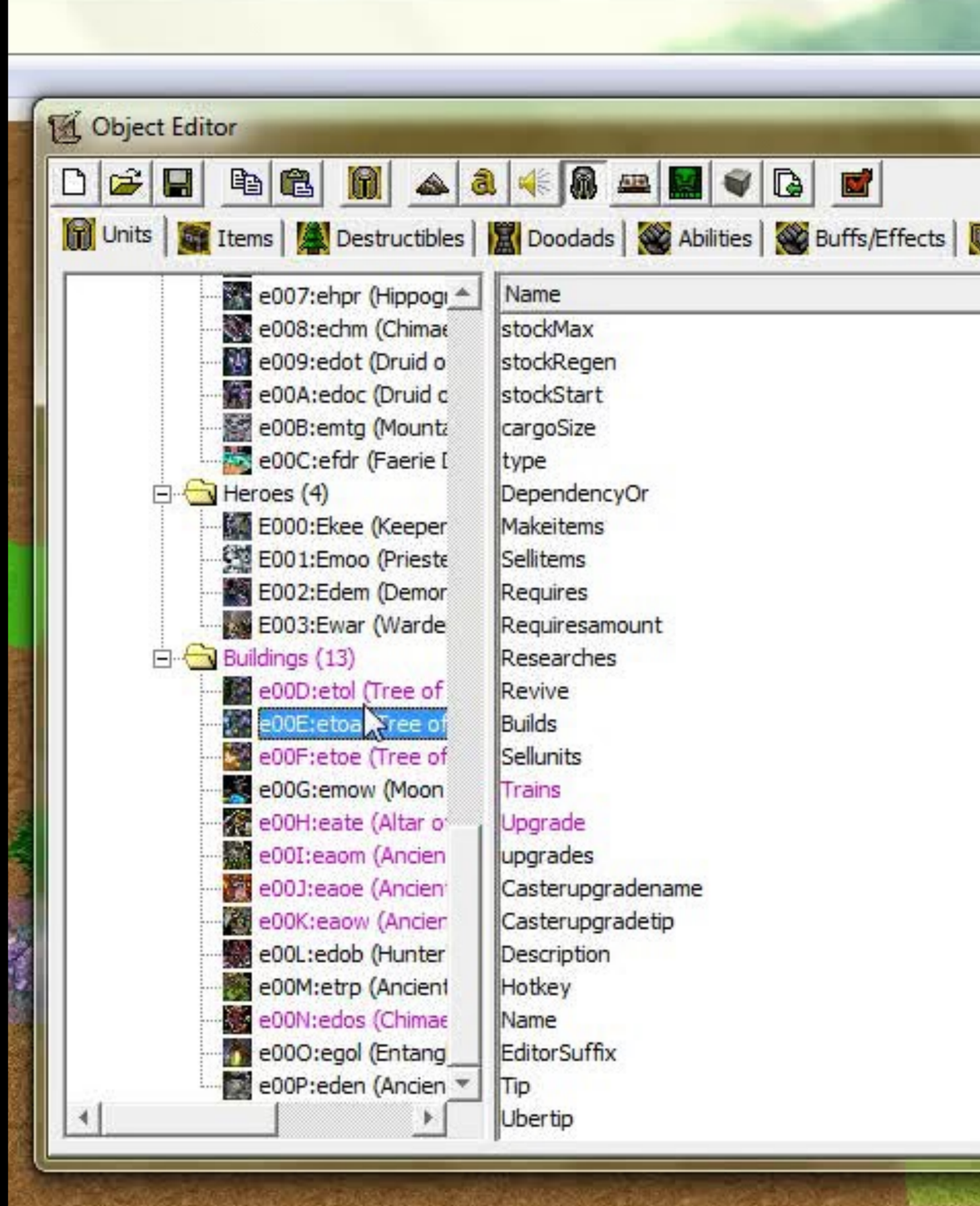

input
message bus
data
update
render
AI
cvafr
quests
any game related logic
Fight system
...
physics
collision detection
animation
cameras
render
UI

Какими свойствами должен обладать движок



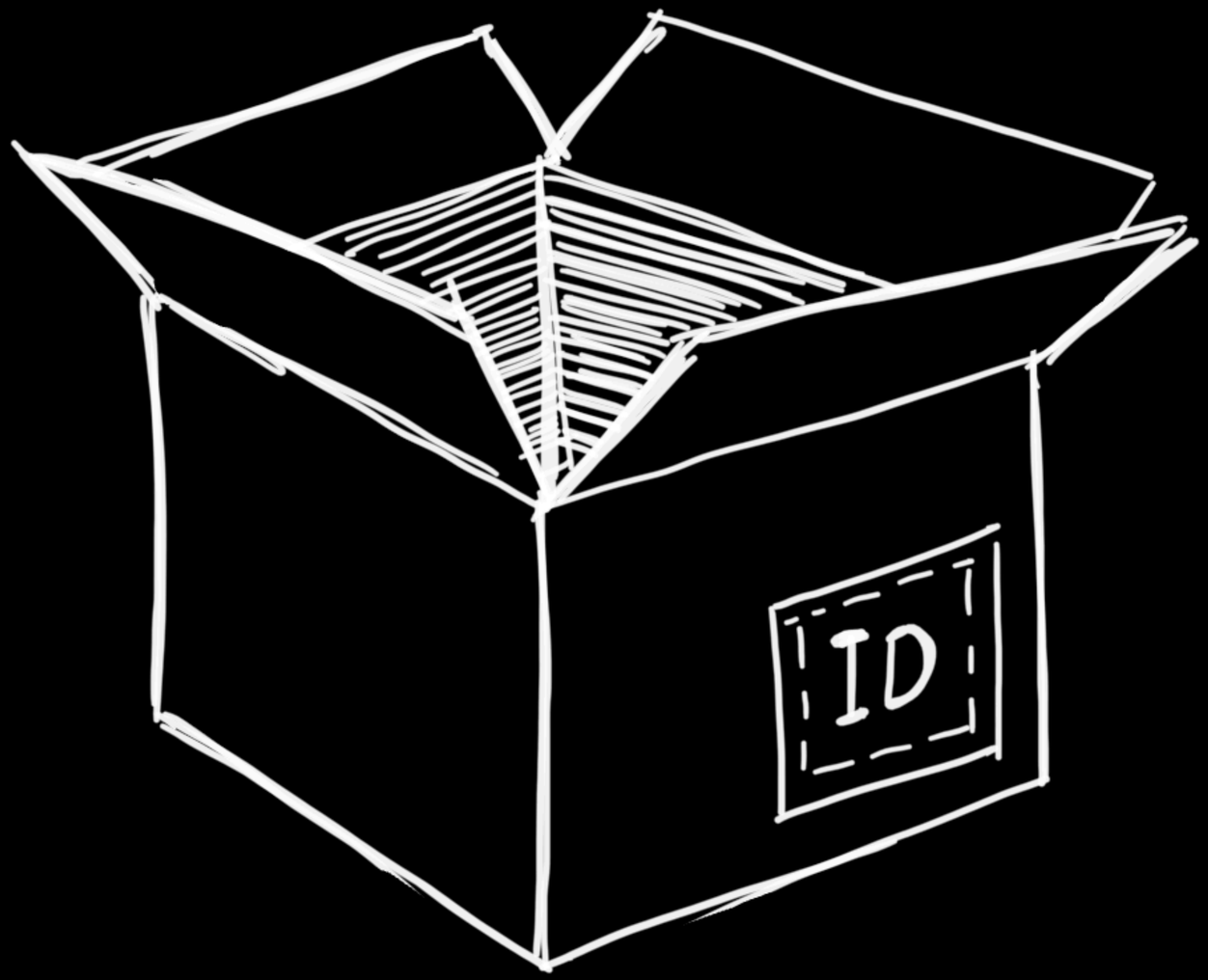
Изолированность

Декларативность



ECS

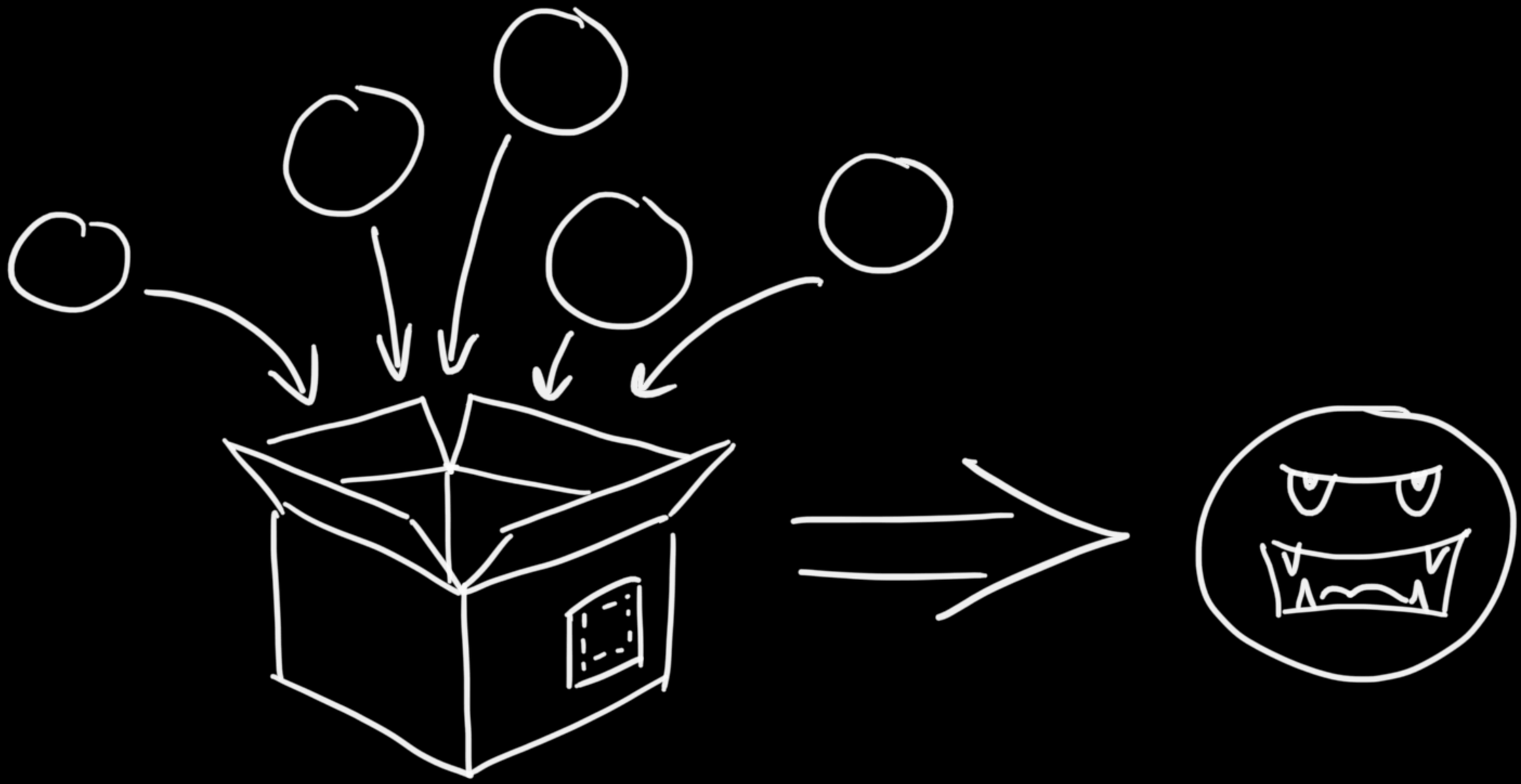
(Entity-Component-System)



Entity

Components


Рисуется
Может Двигаться
Есть Анимация
Вооружен
Управляется ИИ
Может умереть



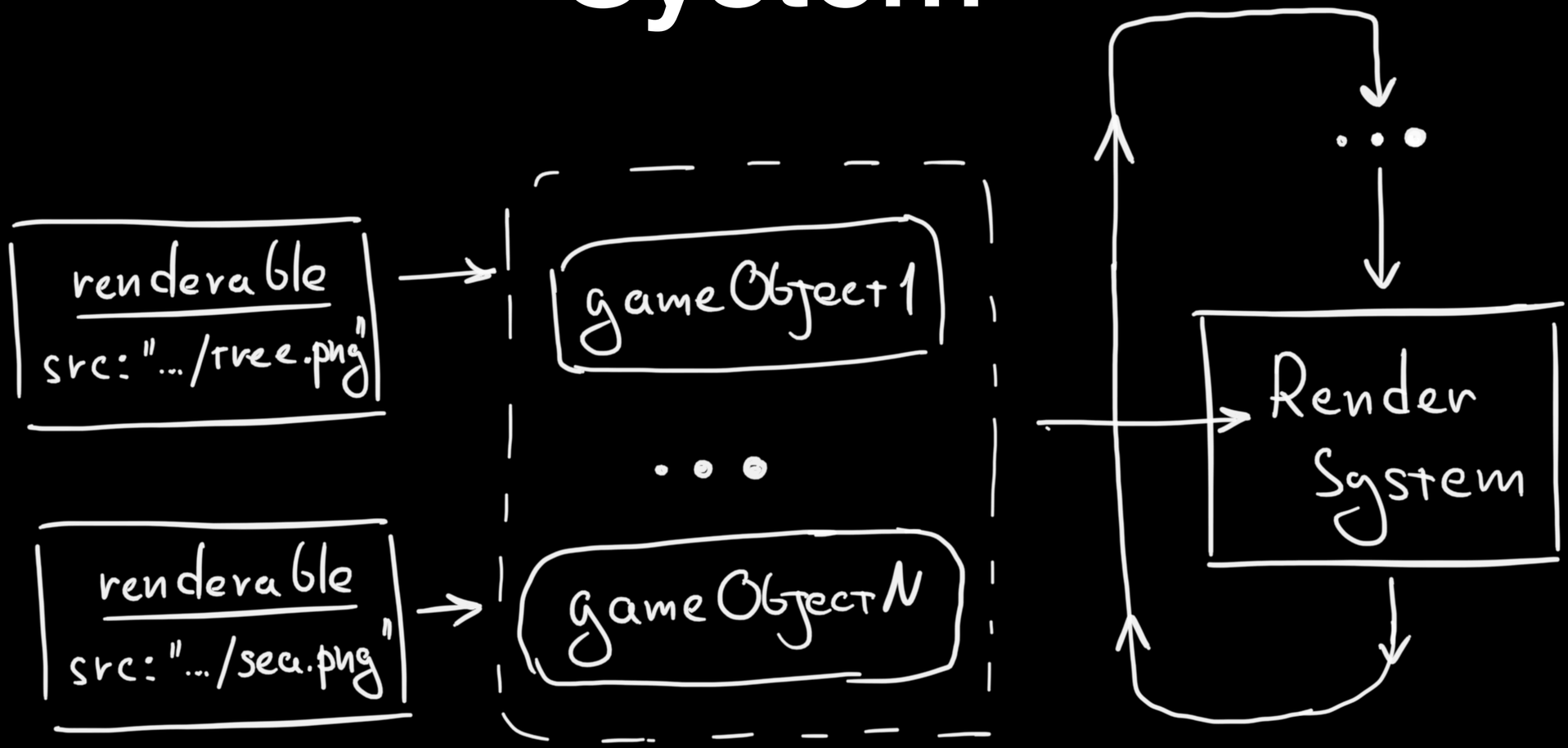
```
{
  "name": "renderable",
  "config": {
    "type": "sprite",
    "src": "units/player",
    "width": 26,
    "height": 26,
    "origin": [ 0, 0 ],
    "flipX": false,
    "flipY": false,
    "sortingLayer": "units"
  }
}
```

Component

Entity

```
class GameObject {
  constructor(id) {}

  getId() {}

  getComponent(name) {}

  setComponent(name, component) {}

  removeComponent(name) {}
}
```

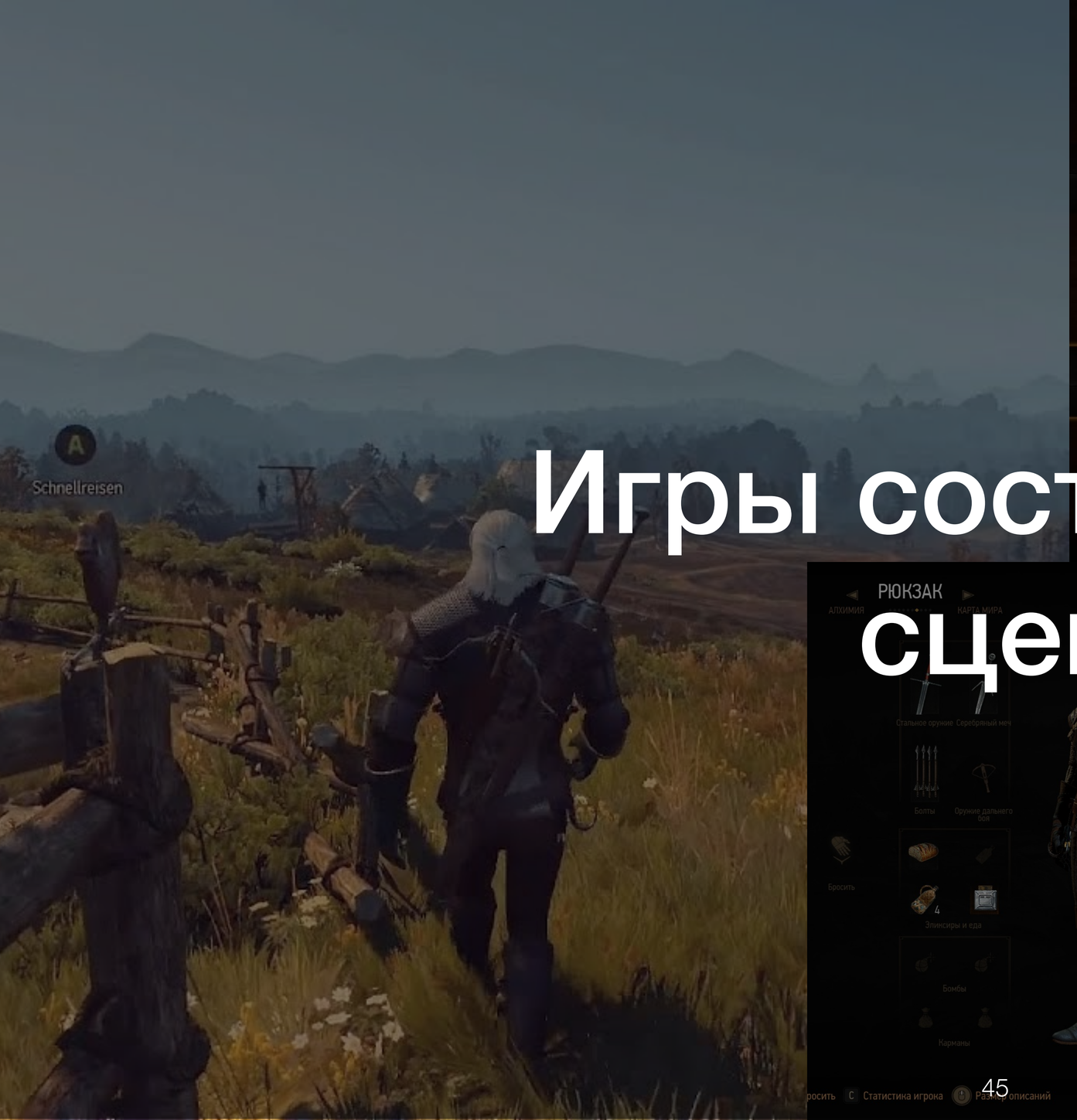
System

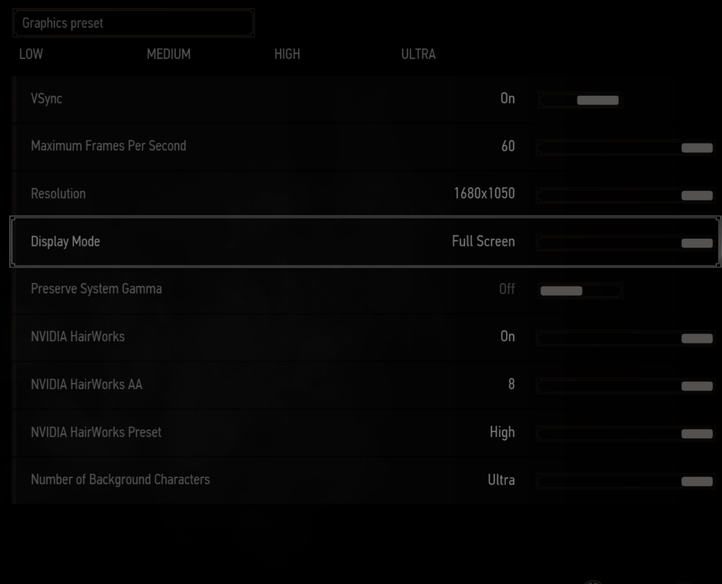

renderable
src: ".../tree.png"
gameObject 1
renderable
src: ".../sea.png"
gameObject N
Render System

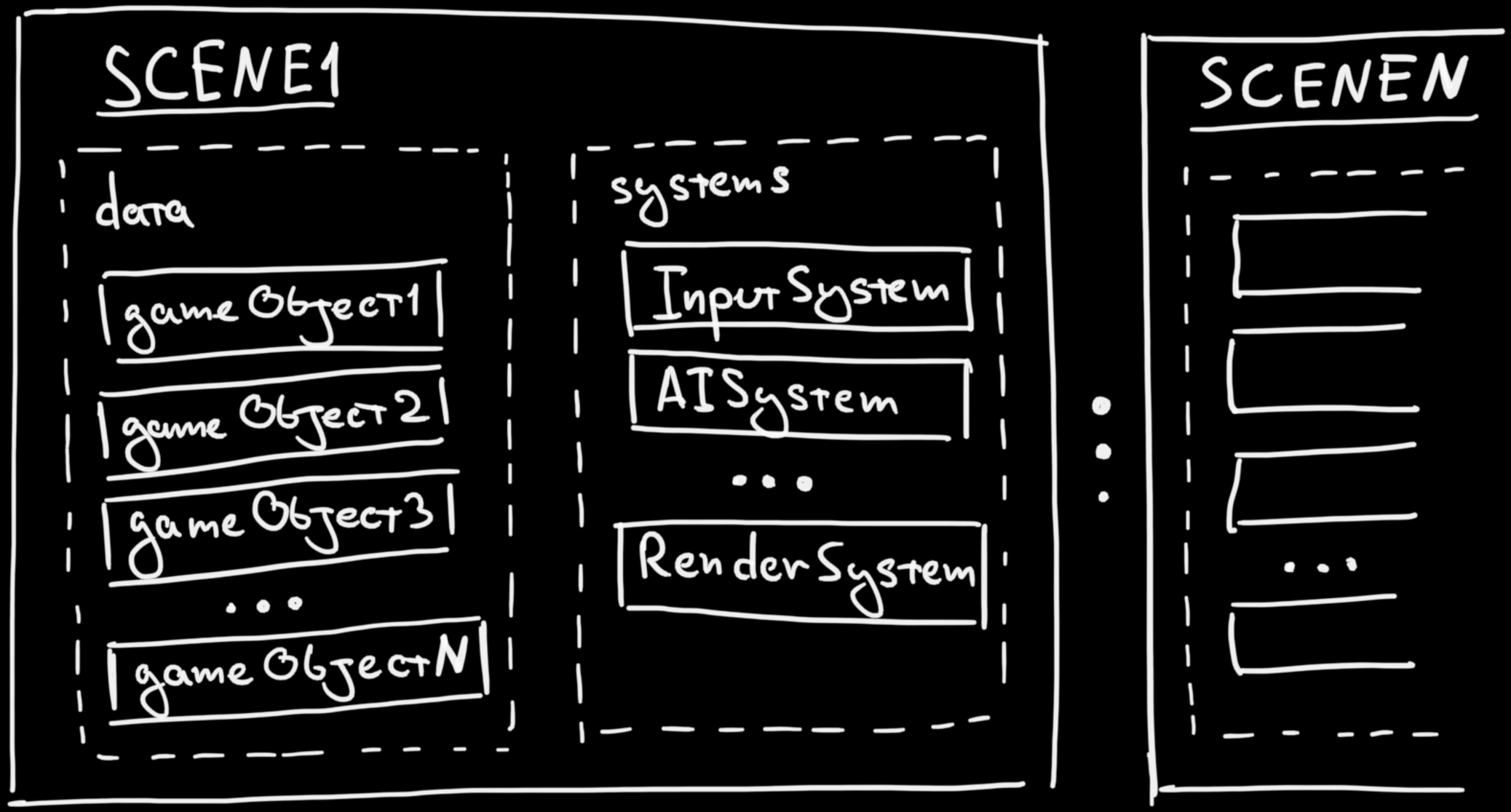
```
class System {
  constructor(gameObjects) {
    this.gameObjects = gameObjects;
  }

  update(deltaTime) {
    this.gameObjects.forEach((gameObject) => {
      /*
       * Здесь мы можем менять позицию объектов
       * const position = gameObject.getComponent('position');
       * position.x += speed * deltaTime;
       *
       * Можем их отрисовать на экране
       * const renderable = gameObject.getComponent('renderable');
       * const position = gameObject.getComponent('position');
       * this.render(renderable, position);
       *
       * ...И так далее
       */
    });
  }
}
```

System

Игры состоят из сцен





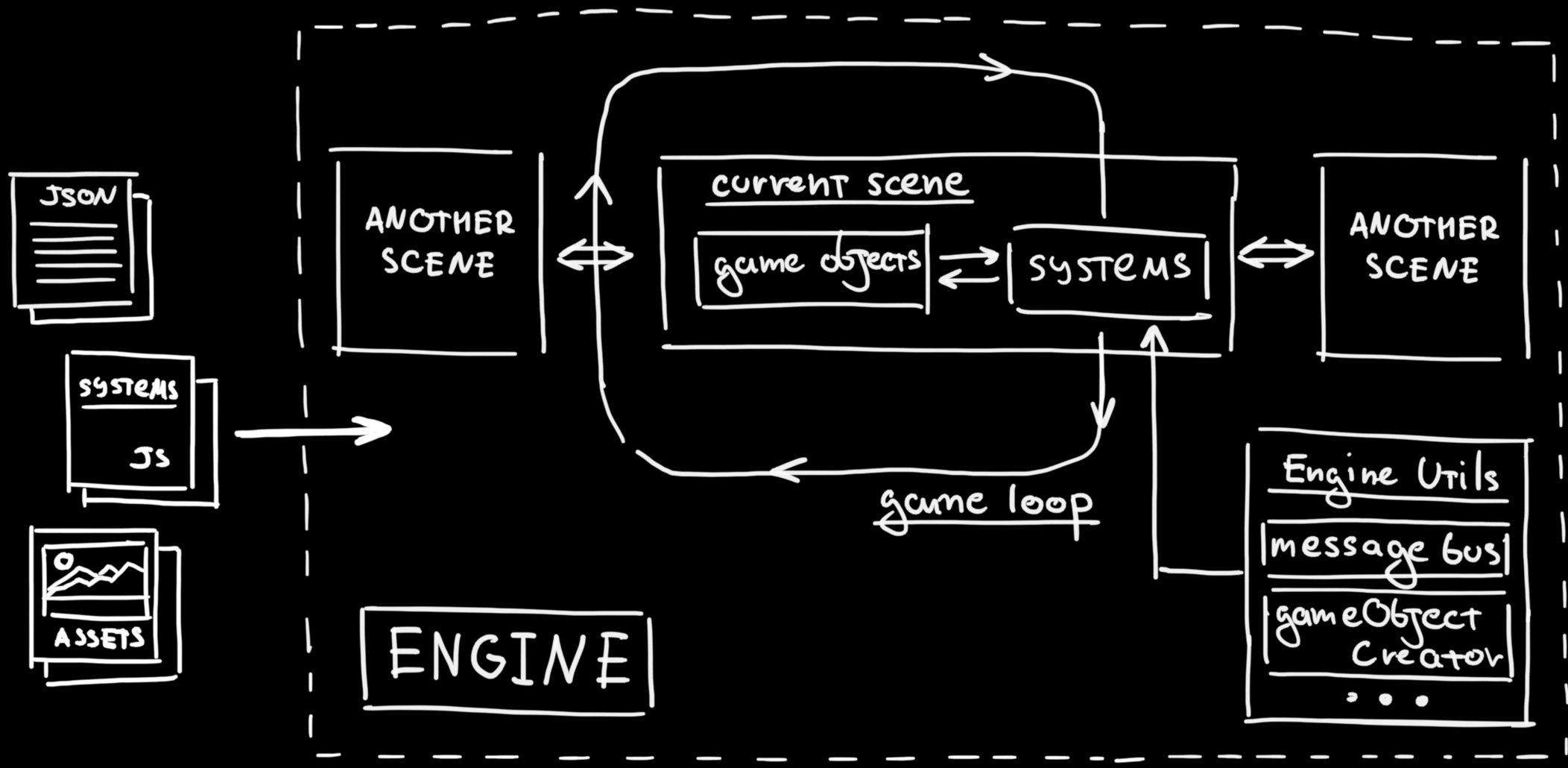

SCENE1
data
gameObject1
gameObject2
gameObject3
...
gameObjectN
systems
InputSystem
AISystem
...
RenderSystem
SCENEN
...

```
{
  "name": "game",
  "gameObjects": [
    {
      "id": "1",
      "name": "player",
      "components": [...]
    },
    ...
    {
      "id": "2",
      "name": "enemy",
      "components": [...]
    }
  ],
  "systems": [
    ...
    {
      "name": "animationSystem",
      "options": {...}
    },
    {
      "name": "renderSystem",
      "options": {...}
    }
  ]
}
```

Сцены описываются в JSON

```
14
15  class Scene {
16    constructor(options) {
17      const { name, gameObjects } = options;
18
19      this.name = name;
20      this.gameObjects = gameObjects.map((config) => new GameObject(config));
21      this.systems = systems.map((config) => new System(
22        config,
23        this.gameObjects,
24        /* Some additional utils */
25        )
26      );
27    }
28
29    getName() {...}
30
31    getSystems() {...}
32
33    getGameObjects() {...}
34
35    addGameObject(gameObject) {...}
36
37    removeGameObject(gameObject) {...}
38  }
39
```

В коде это как-то так


JSON
SYSTEMS
JS
ASSETS
ANOTHER SCENE
current scene
game objects
SYSTEMS
ANOTHER SCENE
game loop
Engine Utils
message bus
gameObject Creator
...
ENGINE


Начнем с визуализации

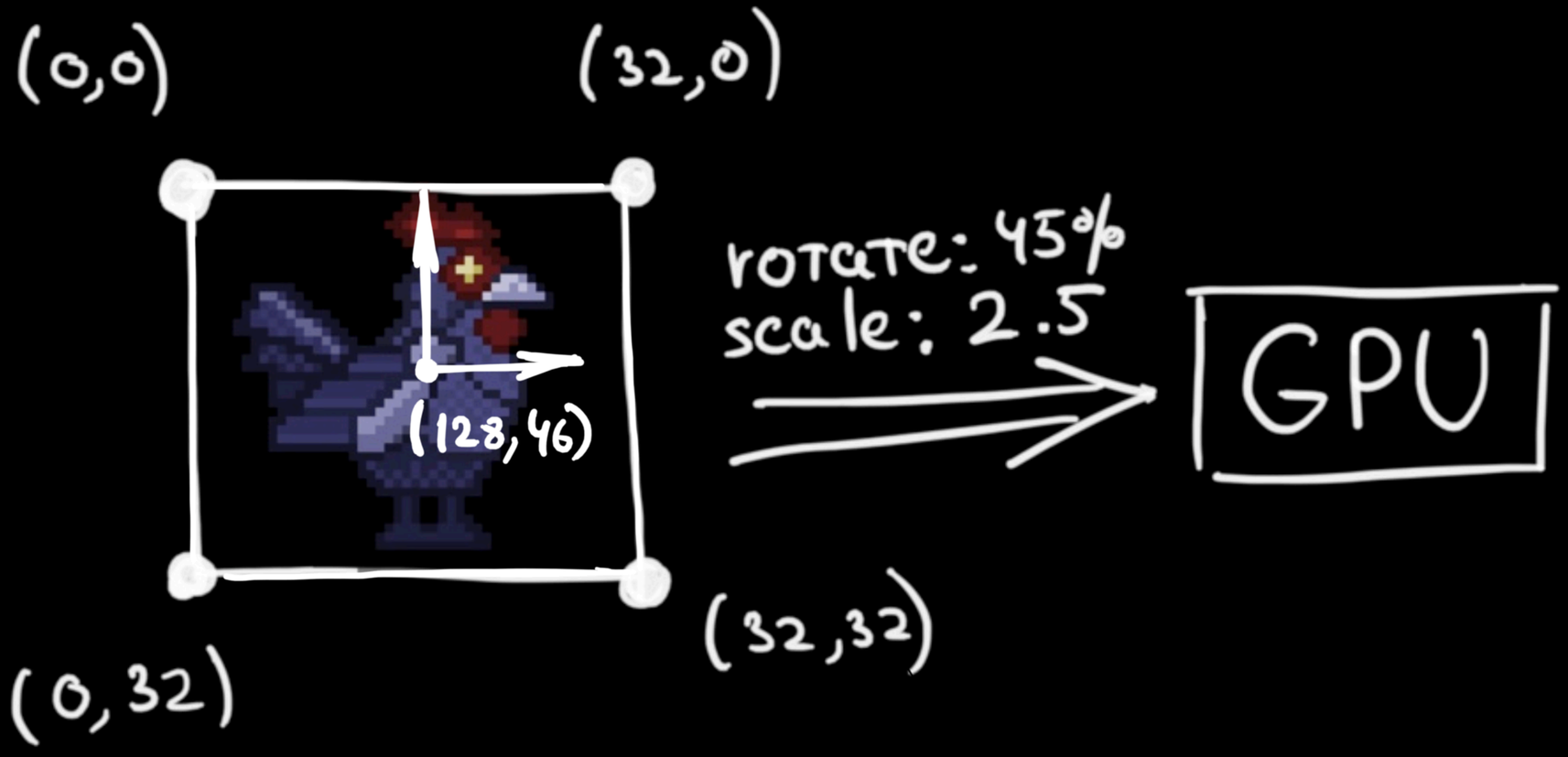
Рисовать можно на чем угодно

- Three.js
- HTML, CSS
- WebGL
- Canvas
- И тд…

Я выбрал

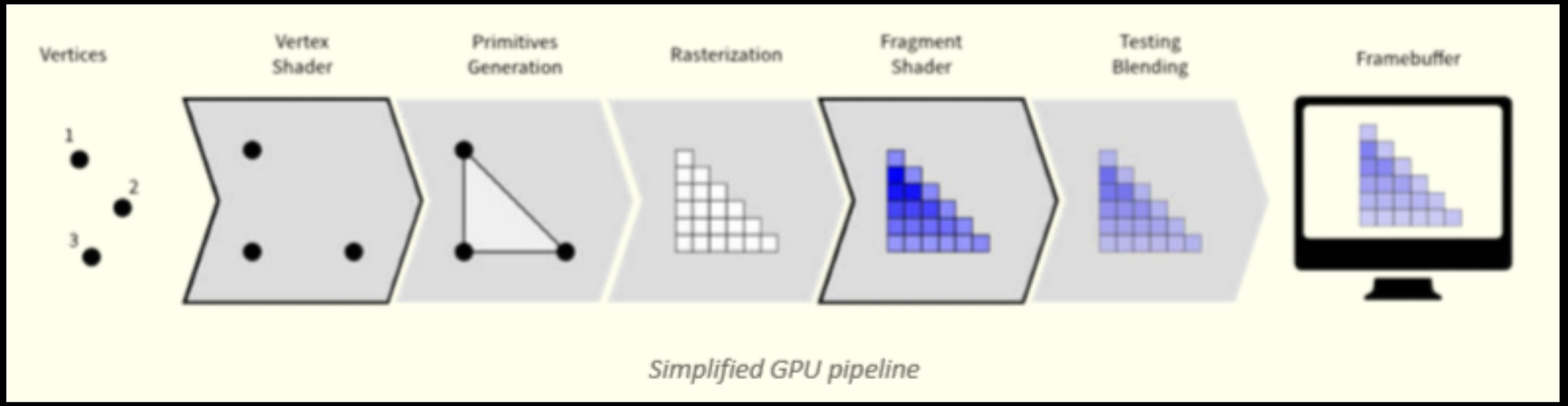
WebGL — мост для передачи данных с CPU на GPU

Вершинный шейдер — вычисляет координаты вершин



```
const vertexShader = glsl`
  // В атрибутах передаются значения для конкретной вершины
  attribute vec2 a_position;
  attribute vec2 a_texCoord;

  // В uniform переменных передаются глобальные параметры
  // в рамках вызова отрисовки
  uniform mat3 u_matrix;

  // Через varying переменные мы передаем данные из вершинного во фрагментный шейдер
  // Во фрагментном шейдере мы получим интерполированные значения
  varying v_texCoord;

  void main() {
    // Специальная переменная вершинного шейдера
    // в которую мы устанавливаем итоговое положение вершины
    gl_Position = vec4((u_matrix * vec3(a_position, 1)).xy, 0, 1);

    // Передаем текстурные координаты в фрагментный шейдер
    v_texCoord = a_texCoord;
  }
`;
```

Шейдеры пишутся на GLSL



Фрагментый шейдер — вычисляет цвет каждого пикселя

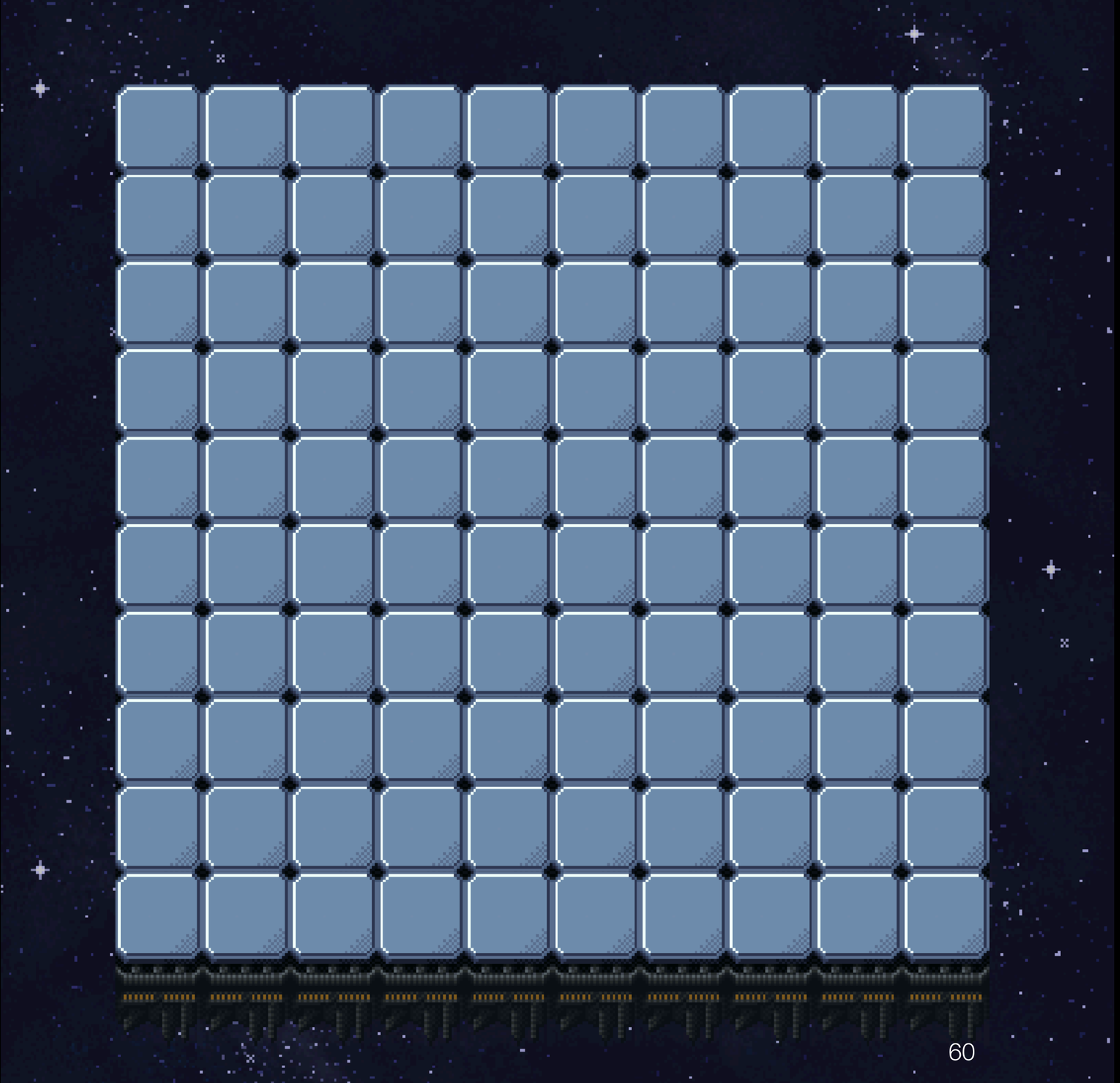
```
const fragmentShader = glsl`
  // Текстура, которую мы передали на видеокарту
  uniform sampler2D u_image;

  // Текстурные координаты, переданные из вершинного шейдера
  varying vec2 v_texCoord;

  void main() {
    // Специальная переменная фрагментного шейдера
    // в которую мы устанавливаем итоговый цвет пикселя
    //
    // В данном случае мы получаем цвет пикселя из текстуры
    // по указанным координатам
    gl_FragColor = texture2D(u_image, v_texCoord);
  }
`;
```

Шейдеры пишутся в строках

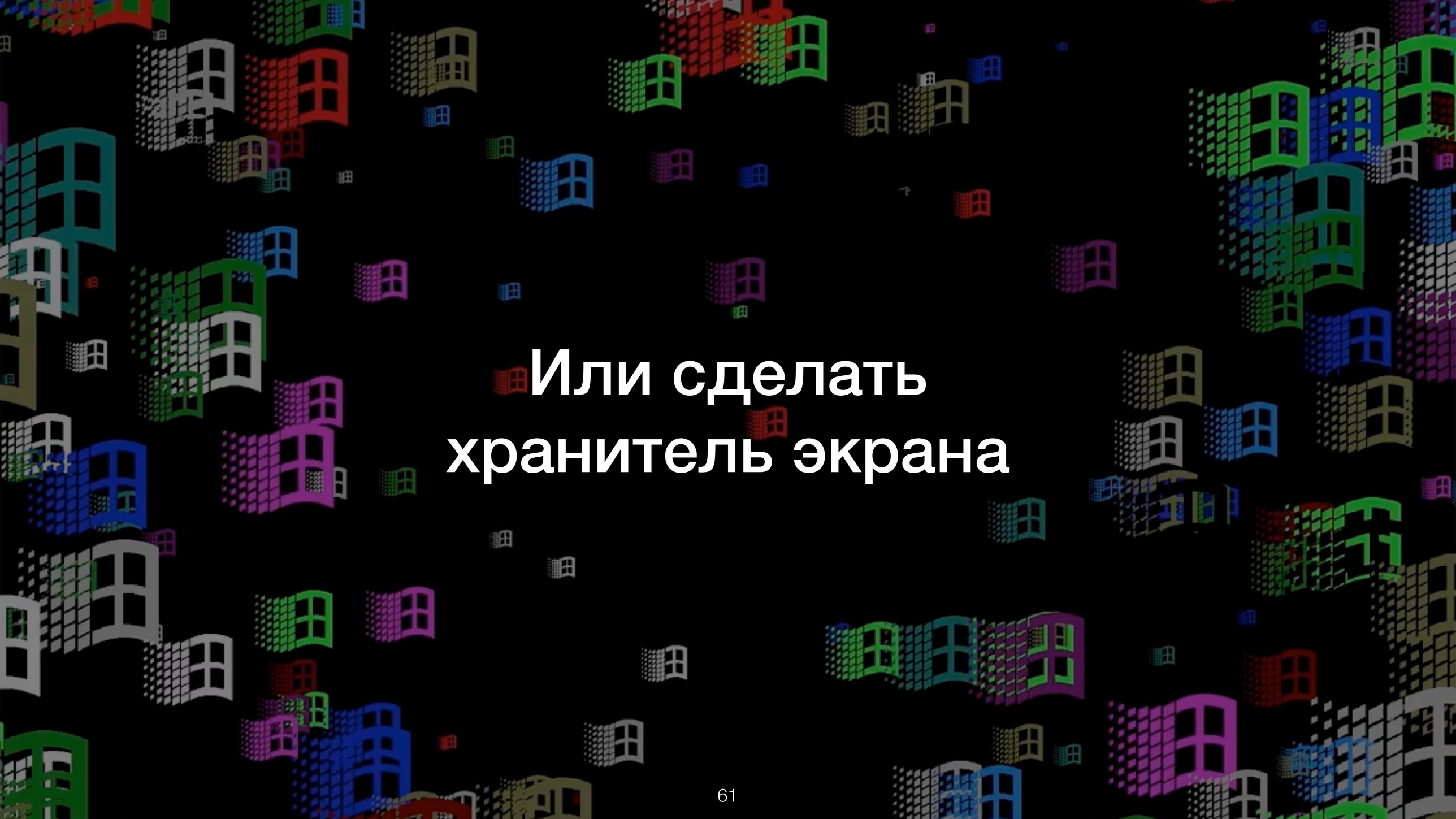
```
class RenderSystem {
  constructor(gameObjects) {
    this.gameObjects = gameObjects;

    this.gl = document.getElementById('root').getContext('webgl');

    /*
     * Компилируем шейдеры
     */
  }

  update() {
    this.gl.clear();

    this.gameObjects.sort();
    this.gameObjects.forEach((gameObject) => {
      /*
       * 1. Задаем буфер с вершинами
       * 2. Задаем uniform переменные (translate, scale, rotate)
       * 3. Задаем текстуру
       */

      this.gl.drawArrays();
    });
  }
}
```

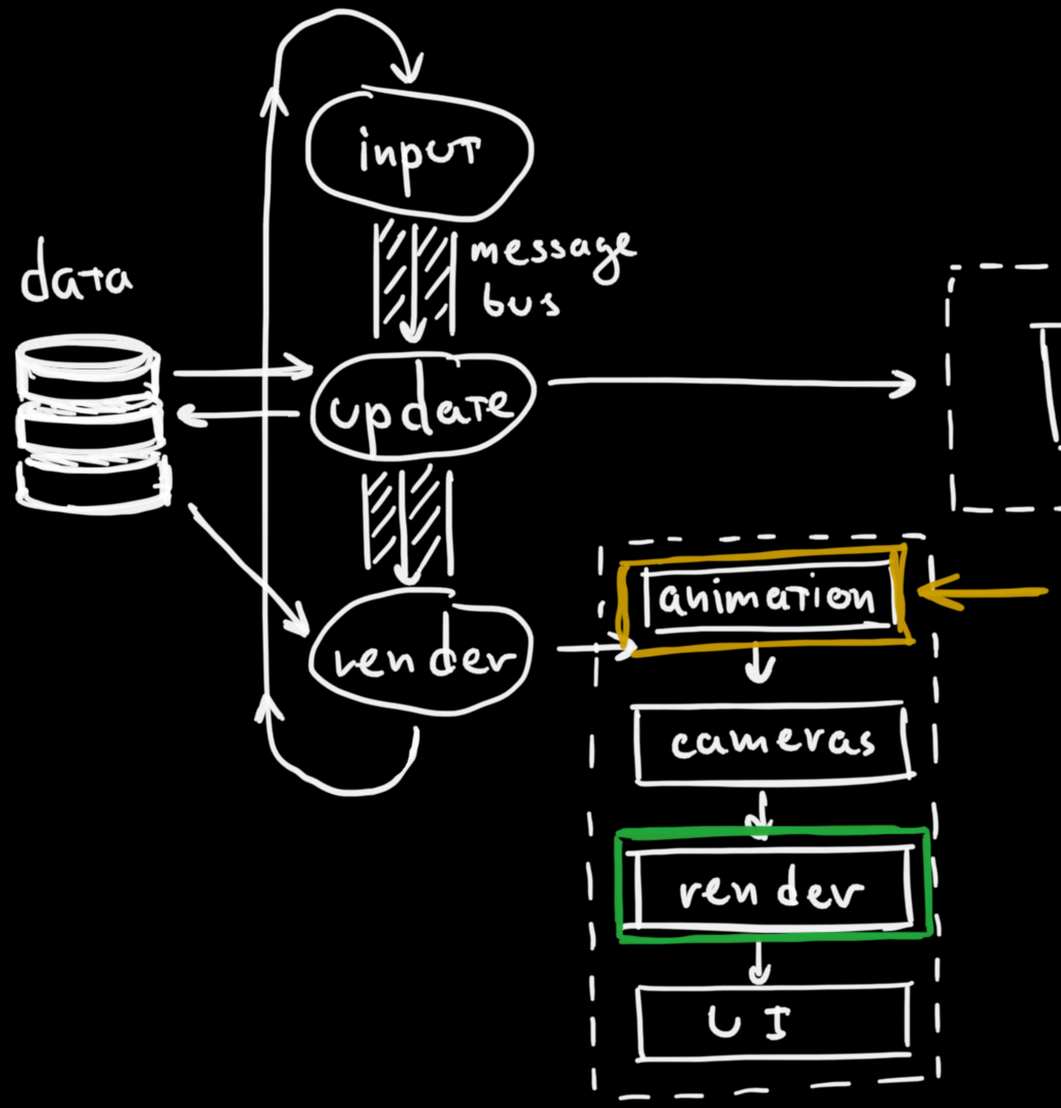
Порядок действий такой

Но тут есть проблемы (может лагать)

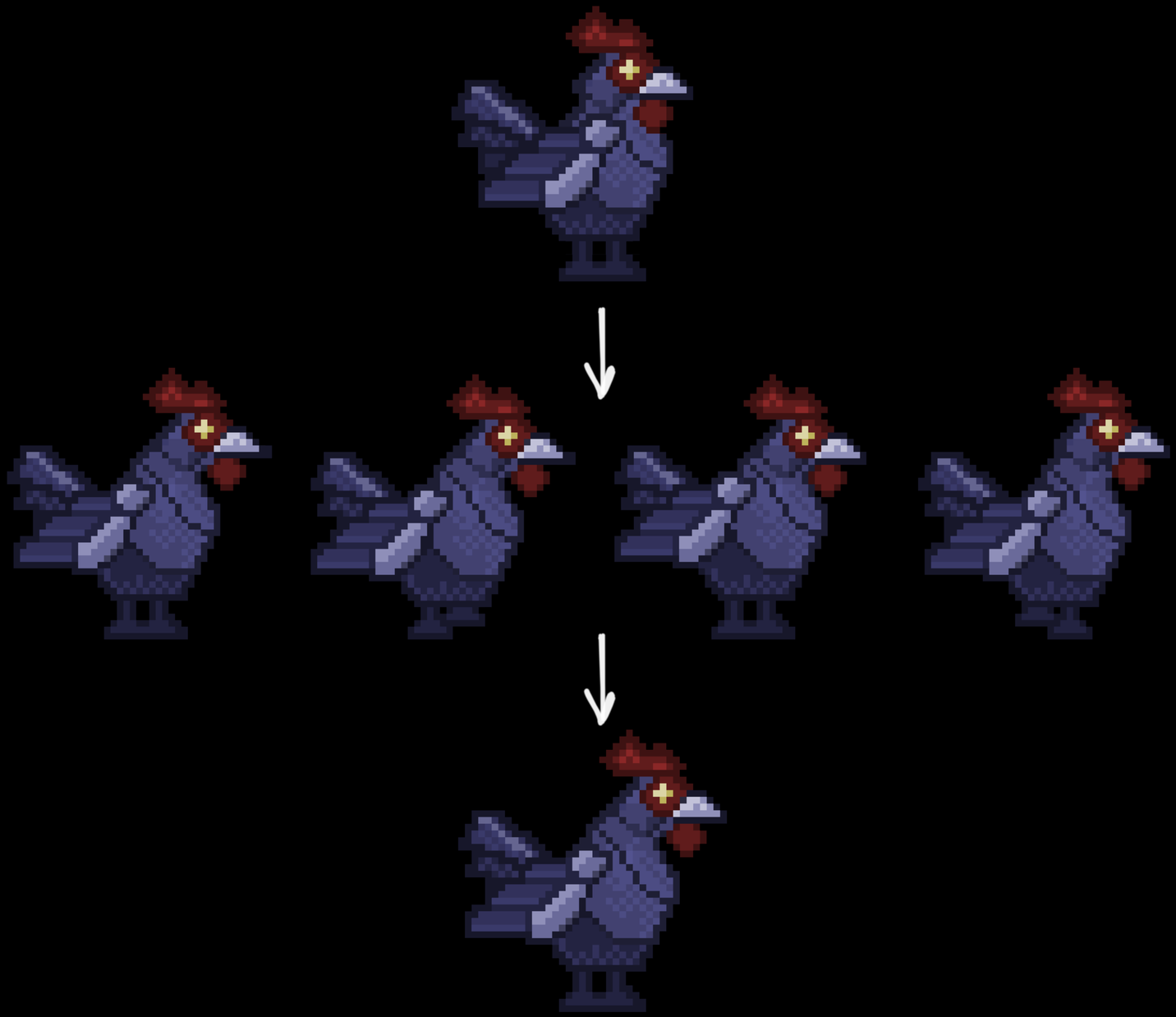


Теперь можно и порисовать
(в JSON-ах)

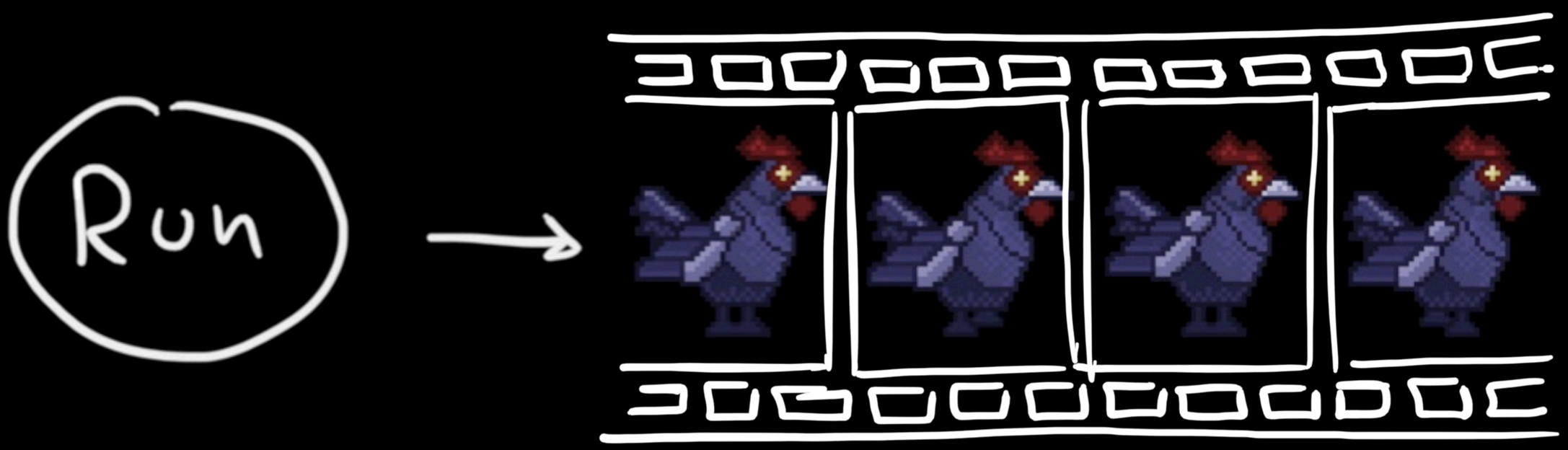

Или сделать
хранитель экрана

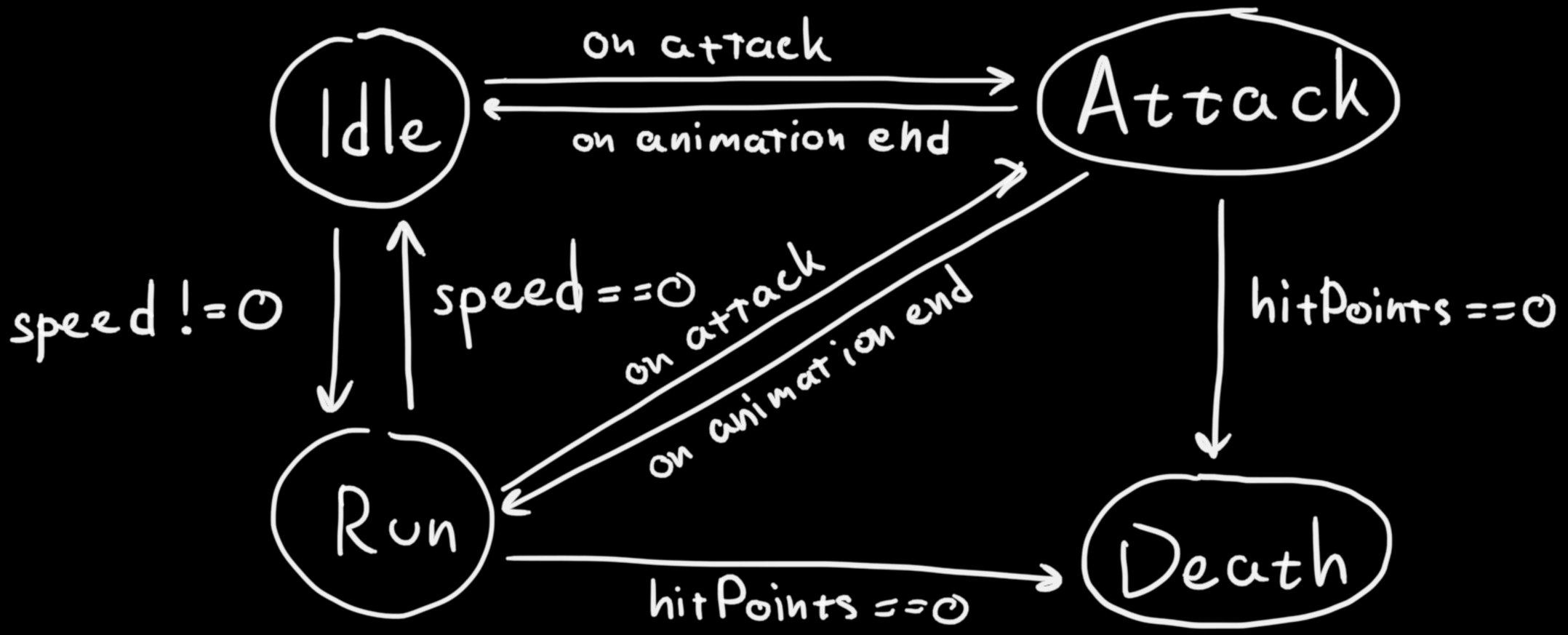

input
message bus
data
update
collision detection
animation
render
cameras
render
UI

Анимация



Анимацию можно представить как машину состояний


Run


Idle
on attack
on animation end
Attack
speed != 0
speed == 0
on attack
on animation end
hitPoints == 0
Run
hitPoints == 0
Death

Учим рендеринг рисовать спрайты

```
{
  "name": "renderable",
  "config": {
    "src": "units/rooster",
    "width": 45,
    "height": 50,
    "sortingLayer": "units"
  }
}
```

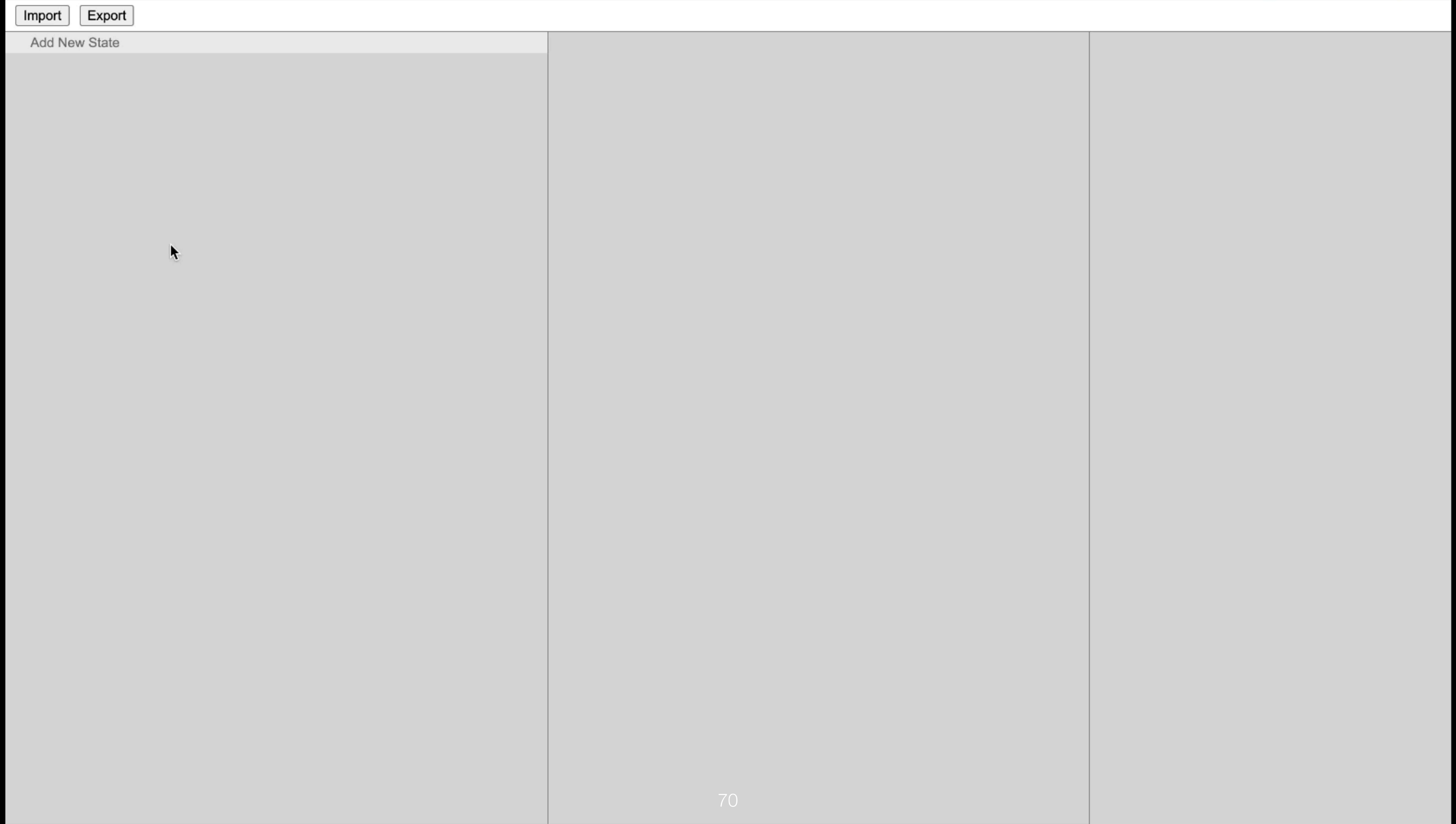
```
{
  "name": "renderable",
  "config": {
    "type": "sprite",
    "src": "units/rooster",
    "width": 45,
    "height": 50,
    "sortingLayer": "units"
  }
}
```

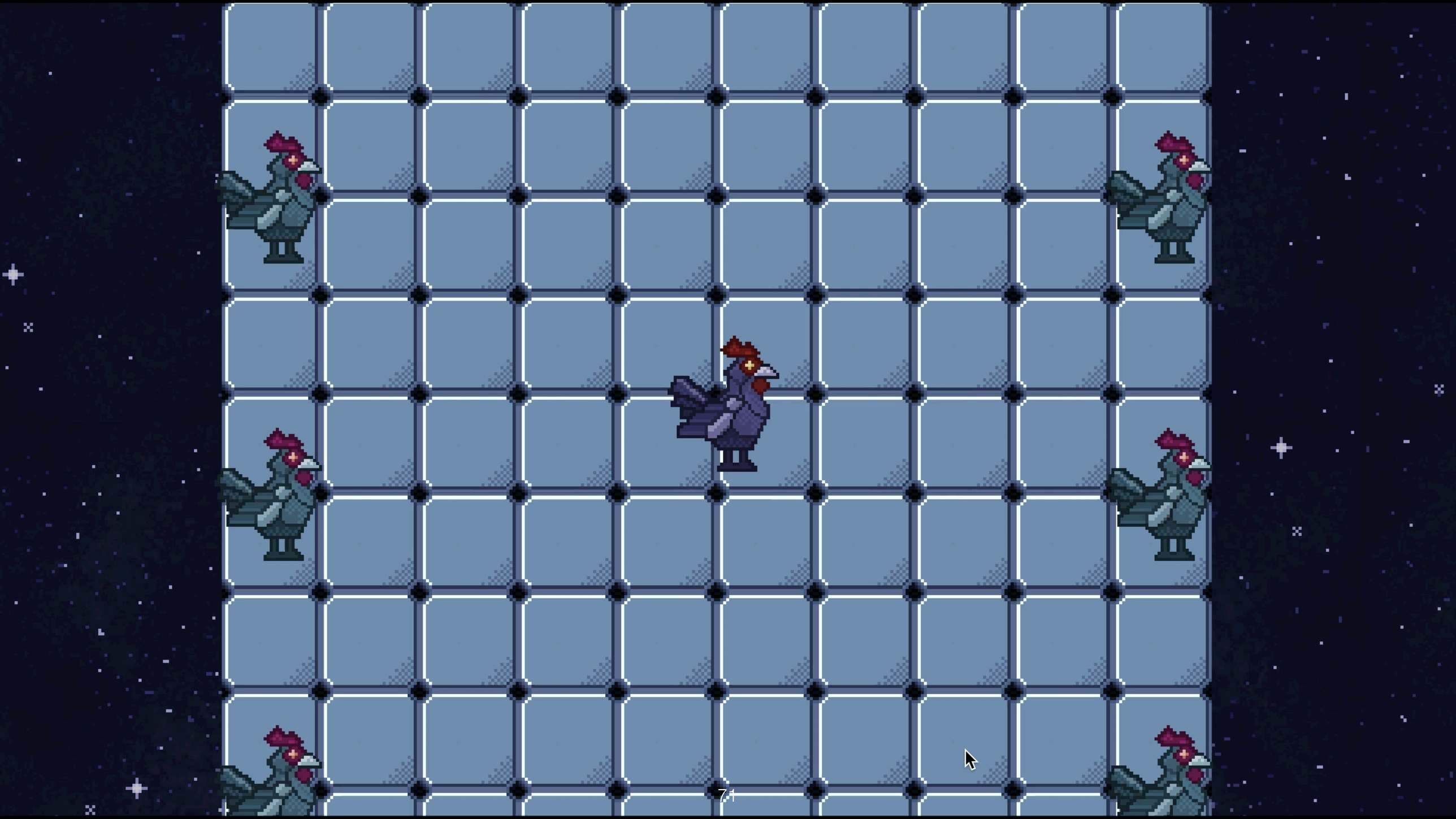
Алгоритм здесь такой

```
class AnimateSystem {
  constructor(gameObjects) {
    this.gameObjects = gameObjects;
  }

  update(options) {
    const deltaTime = options.deltaTime;

    this.gameObjects.forEach((gameObject) => {
      // Вычисляем текущий кадр
      const currentFrame = this.getCurrentFrame(gameObject, deltaTime);
      // Устанавливаем кадр для отрисовки
      this.setFrame(gameObject, currentFrame);

      // Проверяем возможность перехода в другое состояние
      const nextTransition = this.getNextTransition(gameObject);

      if (nextTransition) {
        // Если есть, то устанавливаем новое состояние
        this.setState(gameObject, nextTransition);
      }
    });
  }
}
```

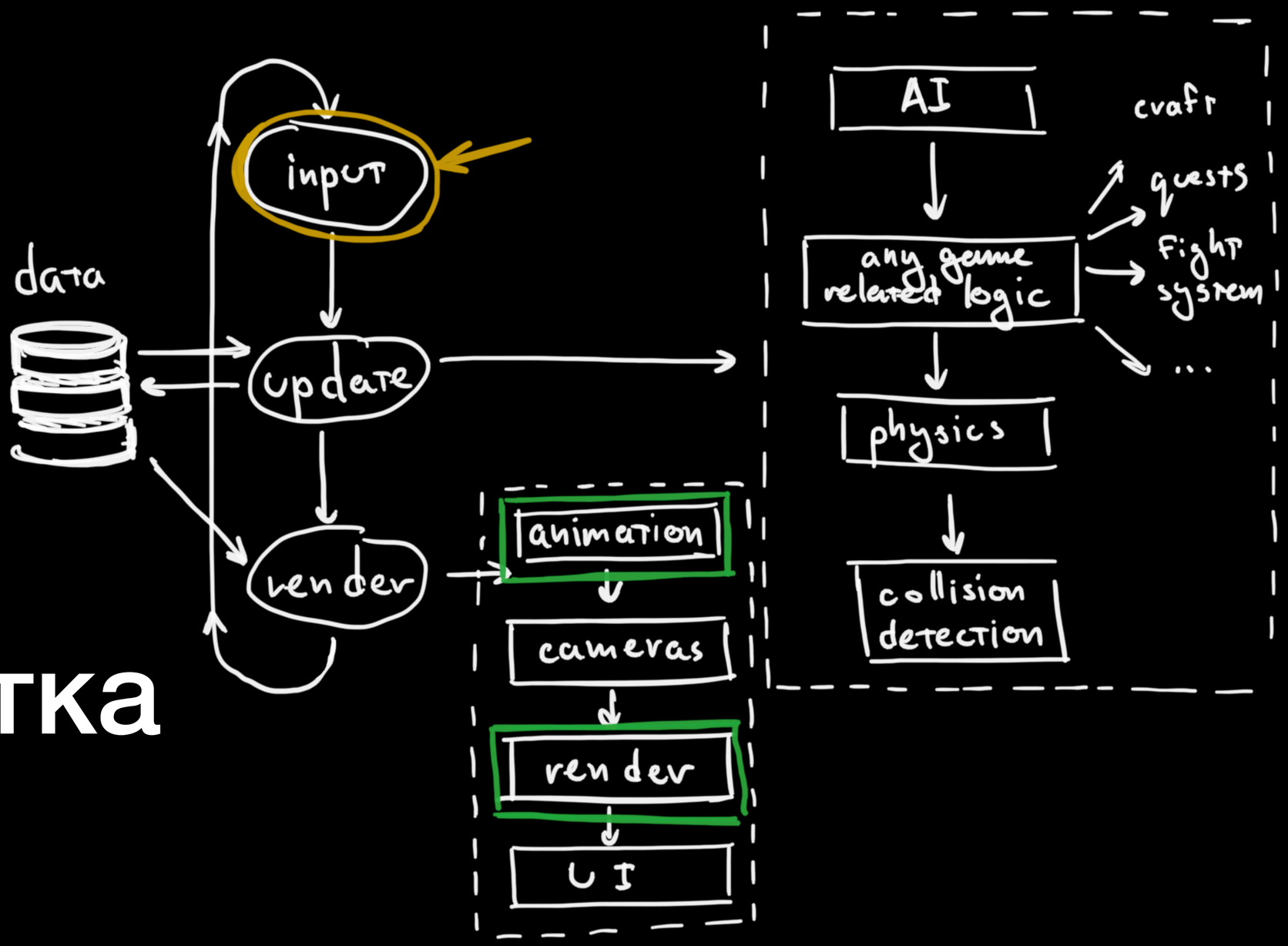
```
{
  "name": "animatable",
  "config": {
    "initialState": "idle",
    "states": [
      {
        "name": "idle",
        "type": "individual",
        "speed": 1,
        "timeline": {...},
        "transitions": [
          {
            "state": "death",
            "conditions": [...]
          }
        ]
      },
      {
        "name": "death",
        "speed": 1,
        "timeline": {
          "frames": [...],
          "looped": false
        },
        "transitions": []
      }
    ]
  }
},
```

Конфиги получаются огромные


Import
Export
Add New State

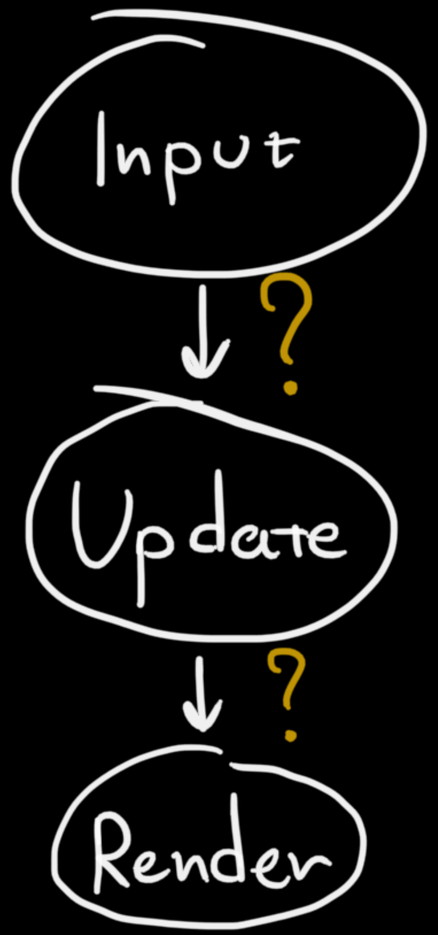


Обработка ввода



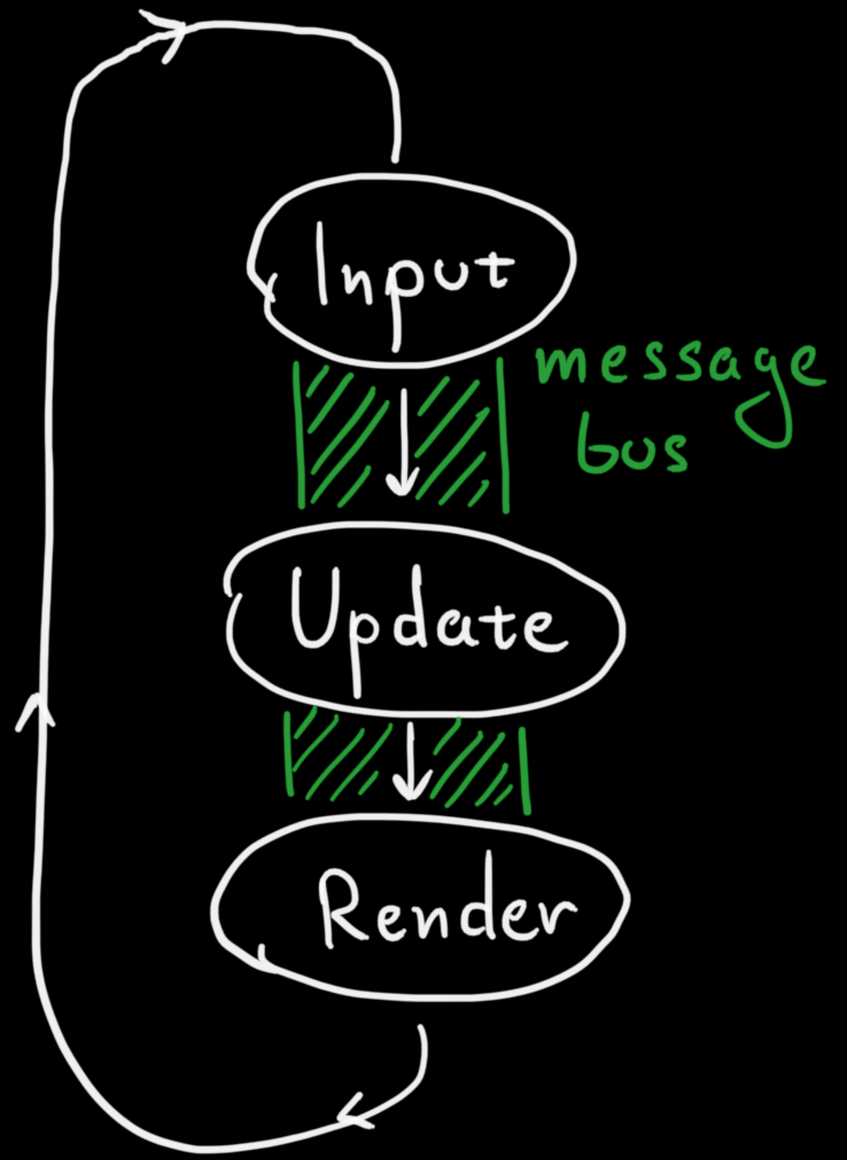
Порядок действий такой

```
class InputSystem {
  constructor() {
    this.inputListener = new InputListener(window);

    // Слушаем события ввода
    this.inputListener.startListen();
  }

  update() {
    // Перебираем накопленную очередь событий
    this.inputListener.getEvents().forEach((event) => {
      // Отправляем событие ввода дальше по игровому циклу
    });

    // Очищаем очередь
    this.inputListener.clear();
  }
}
```



Но как

Message bus



```
class MessageBus {
  constructor() {
    this._messages = {};
  }

  send(message) {}

  get(messageType) {}

  delete(messageType) {}

  clear() {}
}
```

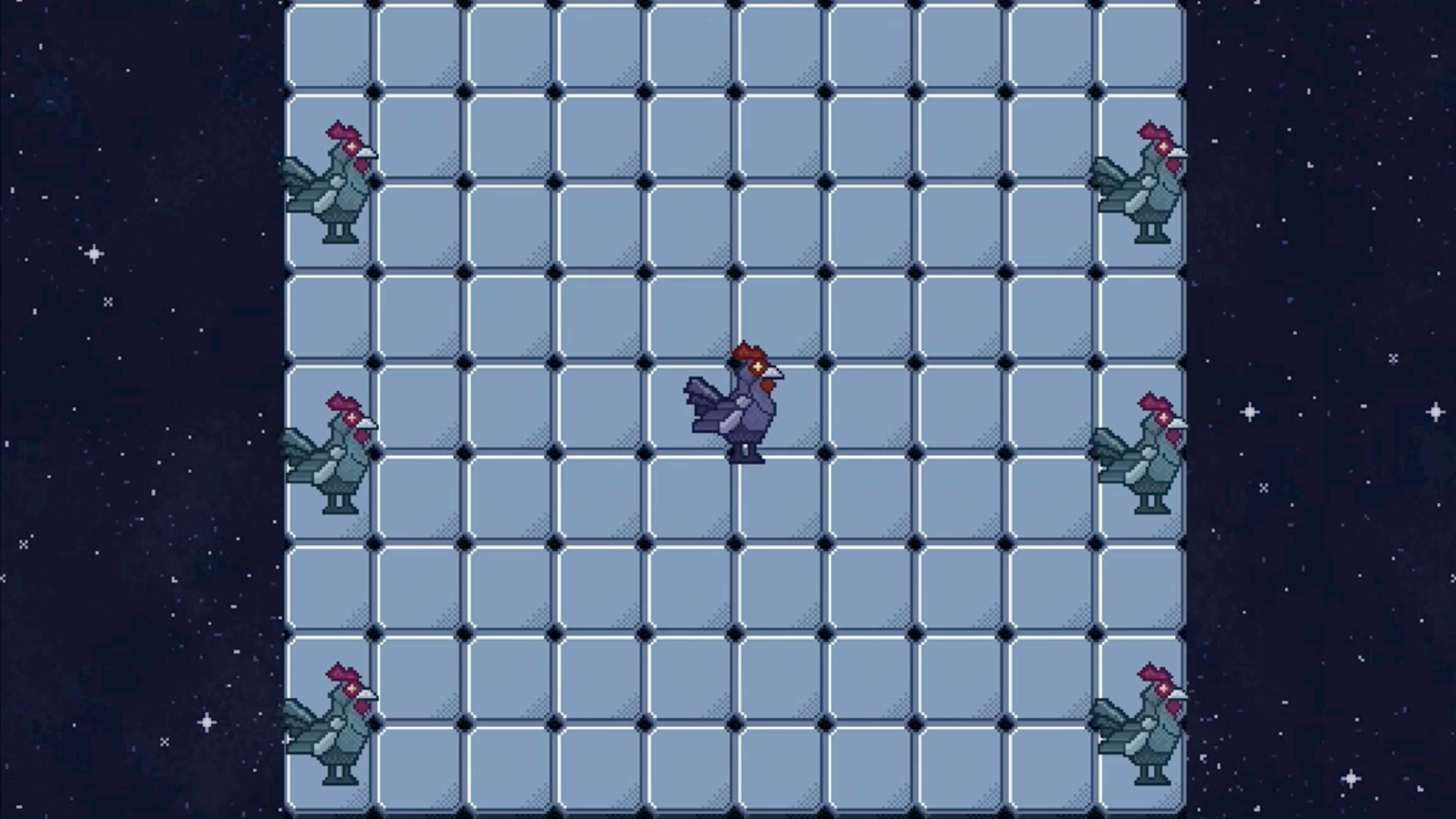
Message bus

Системы

```
/* Слушаем события ввода и
пересылаем их в движок */
{
  "name": "keyboardInputSystem"
},

/* Маппинг клавиш на действия
(движение, прыжок, атака, …) */
{
  "name": "keyboardControlSystem"
},

/* Двигаем все объекты с
keyboardControl */
{
  "name": "movementSystem"
},
```

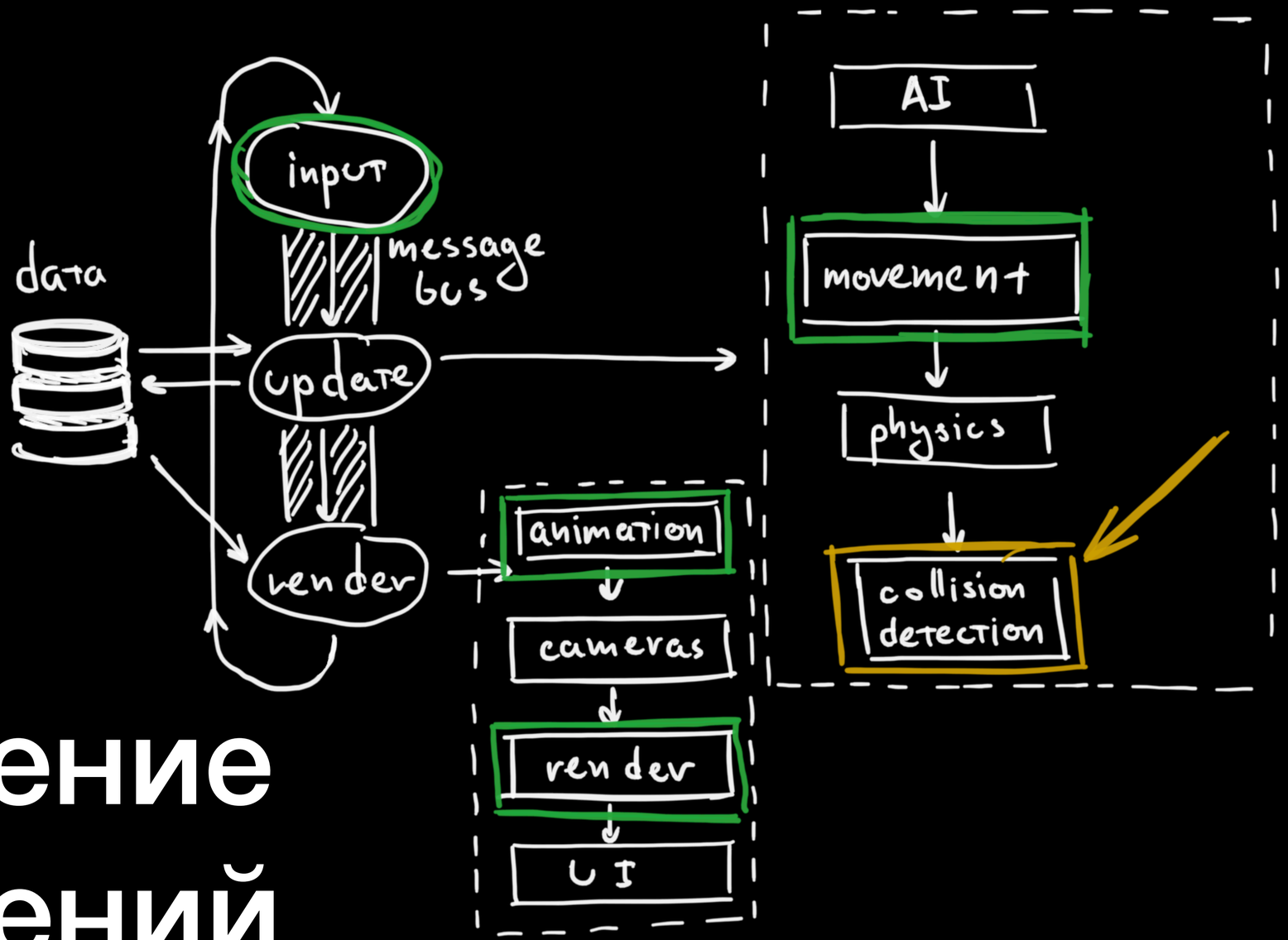
Компоненты

```
{
  "name": "keyboardControl",
  "config": {
    "inputEventBindings": {
      "W_PRESSED": {...},
      "A_PRESSED": {...},
      "S_PRESSED": {...},
      "D_PRESSED": {...}
    }
  }
},
{
  "name": "movement",
  "config": {
    "speed": 100
  }
},
```



Но получилось не сразу

Обнаружение столкновений



Для чего это вообще нужно

Блокировать путь через препятствия

Регистрировать попадание снаряда по цели

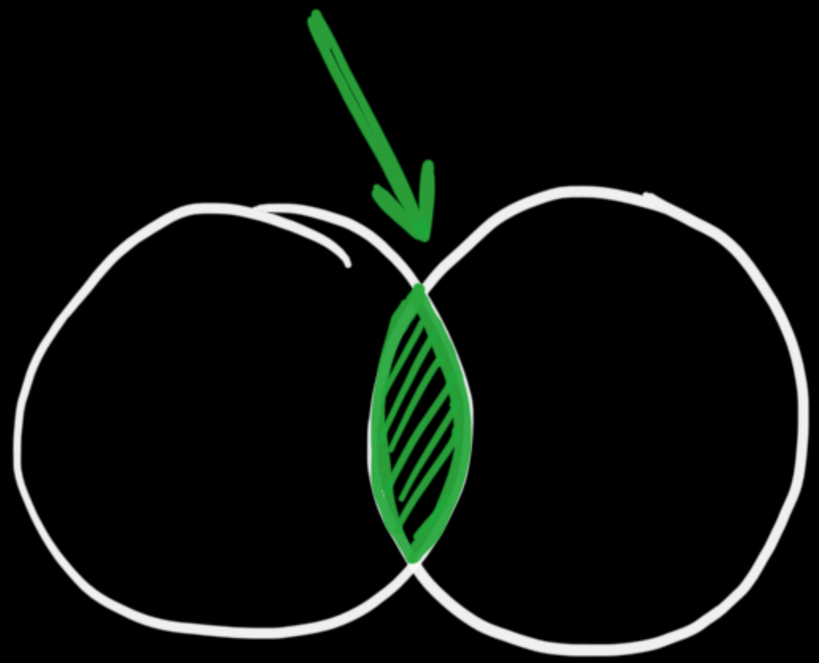
Подбирать предметы

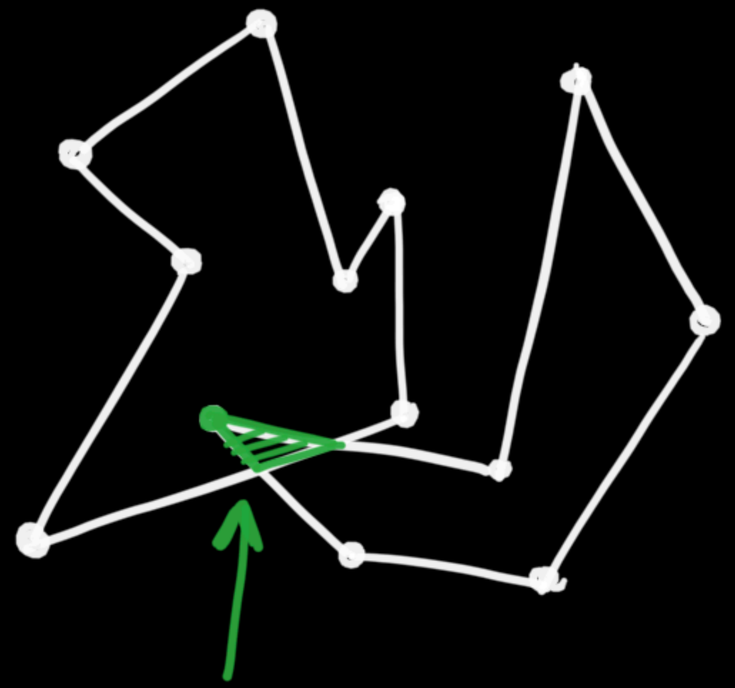
И многое другое

Обычно реальные габариты персонажа вот такие

А не такие

Потому что обнаружить это проще





Чем это

А разницы никто и не заметит

Colliders

```
{
  "name": "colliderContainer",
  "config": {
    "type": "boxCollider",
    "collider": {
      "sizeX": 33,
      "sizeY": 40,
      "centerX": 0,
      "centerY": 0
    }
  }
}
```

```
{
  "name": "colliderContainer",
  "config": {
    "type": "circleCollider",
    "collider": {
      "radius": 10,
      "centerX": 0,
      "centerY": 0
    }
  }
},
```

```
this.gameObjects.forEach((gameObject1) => {
  this.gameObjects.forEach((gameObject2) => {
    if (gameObject1.getId() !== gameObject2.getId()) {
      if (this.checkOnIntersection(gameObject1, gameObject2)) {
        this.messageBus({
          /* Some message about collision */
        });
      }
    }
  });
});
```

```
this.gameObjects.forEach((gameObject1) => {
  this.gameObjects.forEach((gameObject2) => {
    if (gameObject1.getId() !== gameObject2.getId()) {
      if (this.checkOnIntersection(gameObject1, gameObject2)) {
        this.messageBus({
          /* Some message about collision */
        });
      }
    }
  });
});
```

O(n^2)

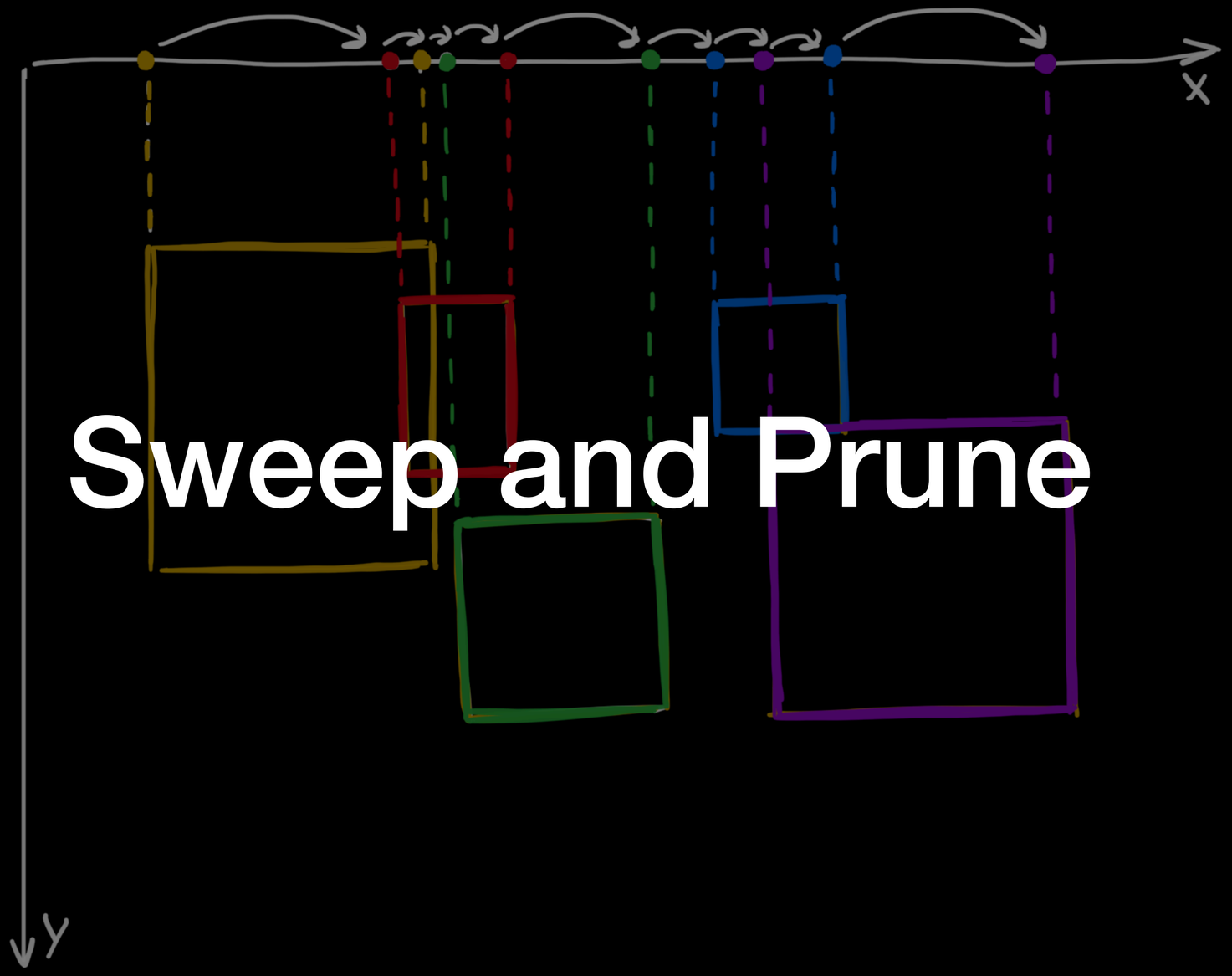
Две фазы

Широкая

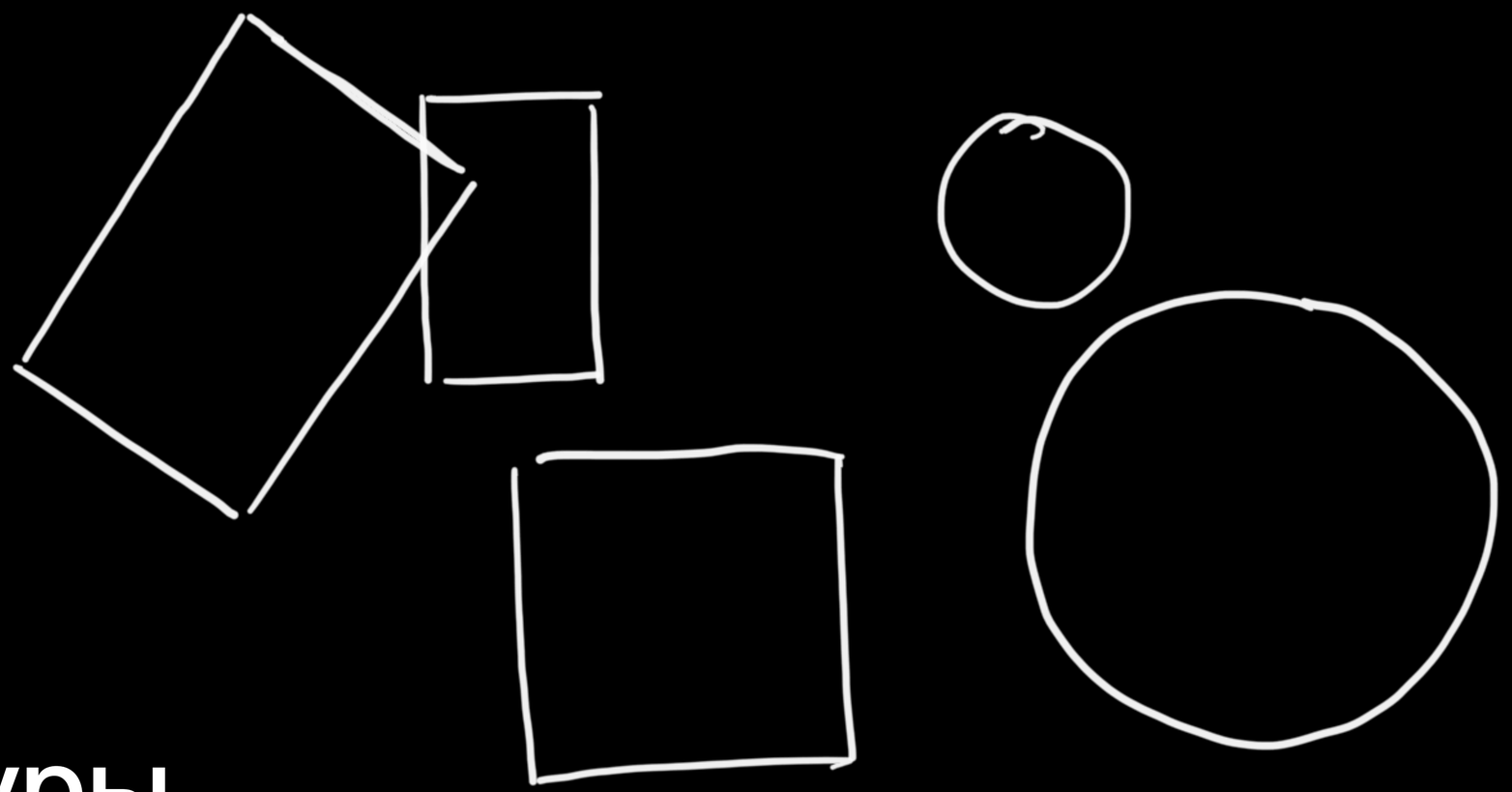
Грубо отсеиваем пары, которые точно не пересекаются

Узкая

Более тяжелый, но точный тест.
Помимо факта пересечения дает доп. инфу (глубина проникновения, …)

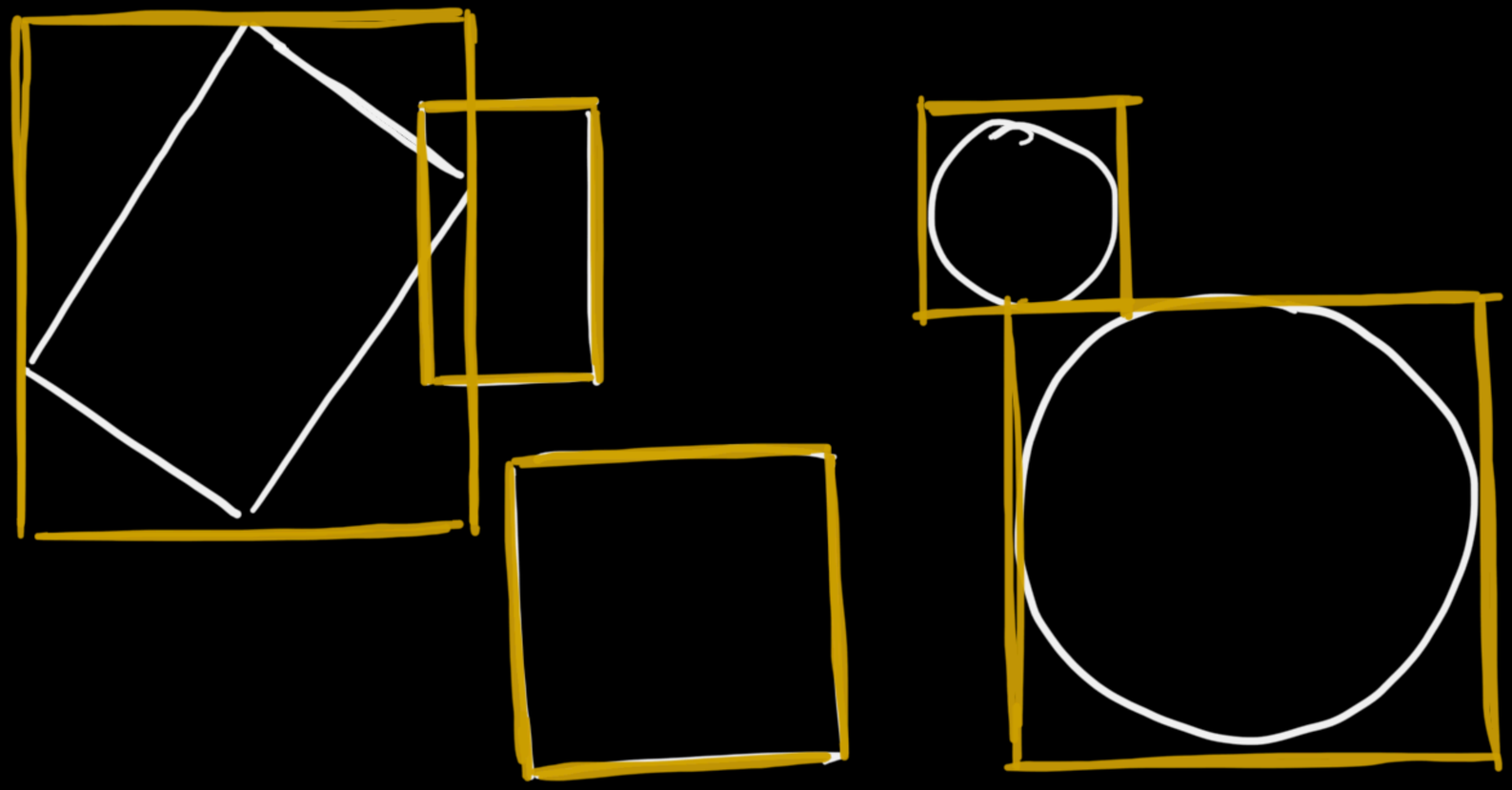


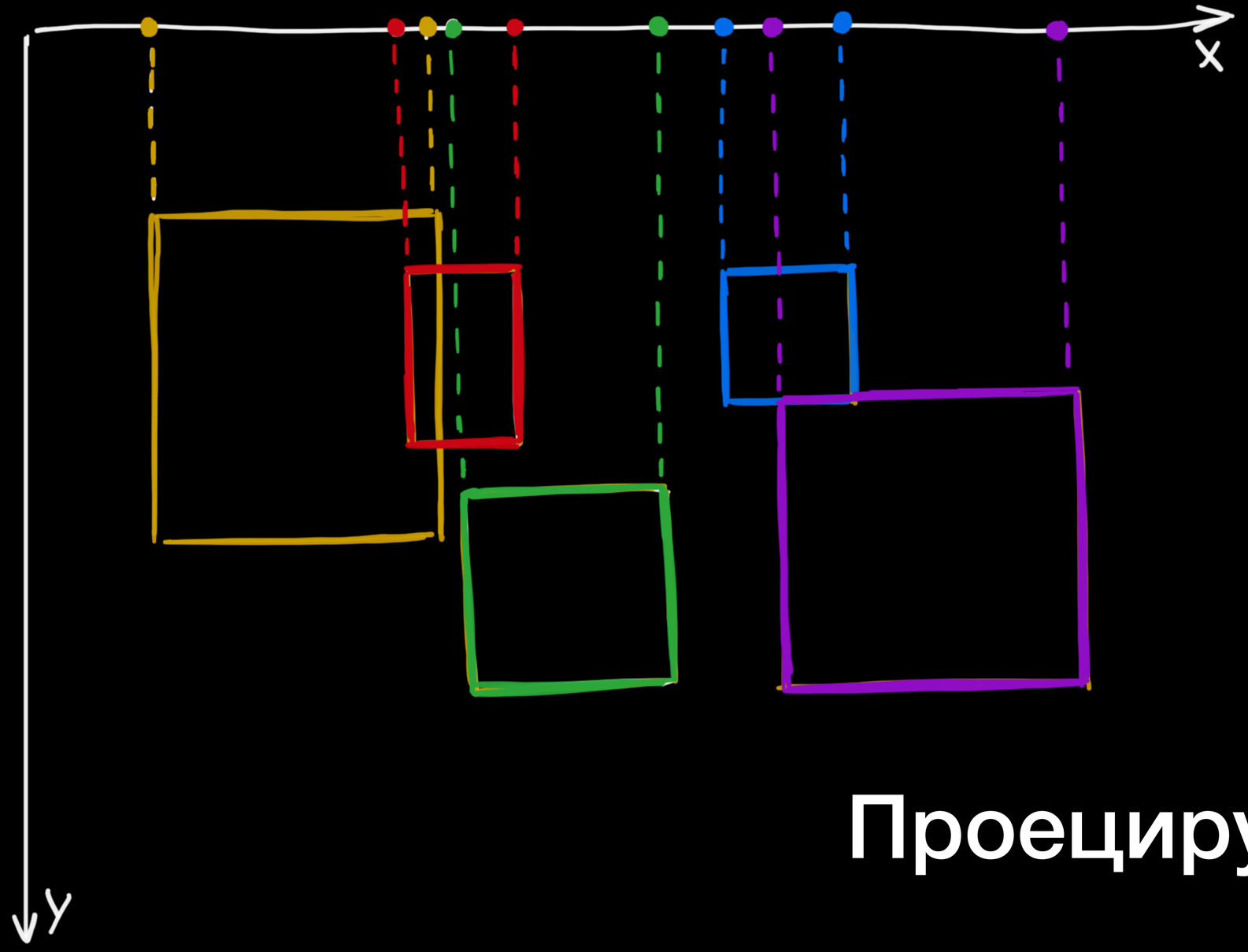
Sweep and Prune



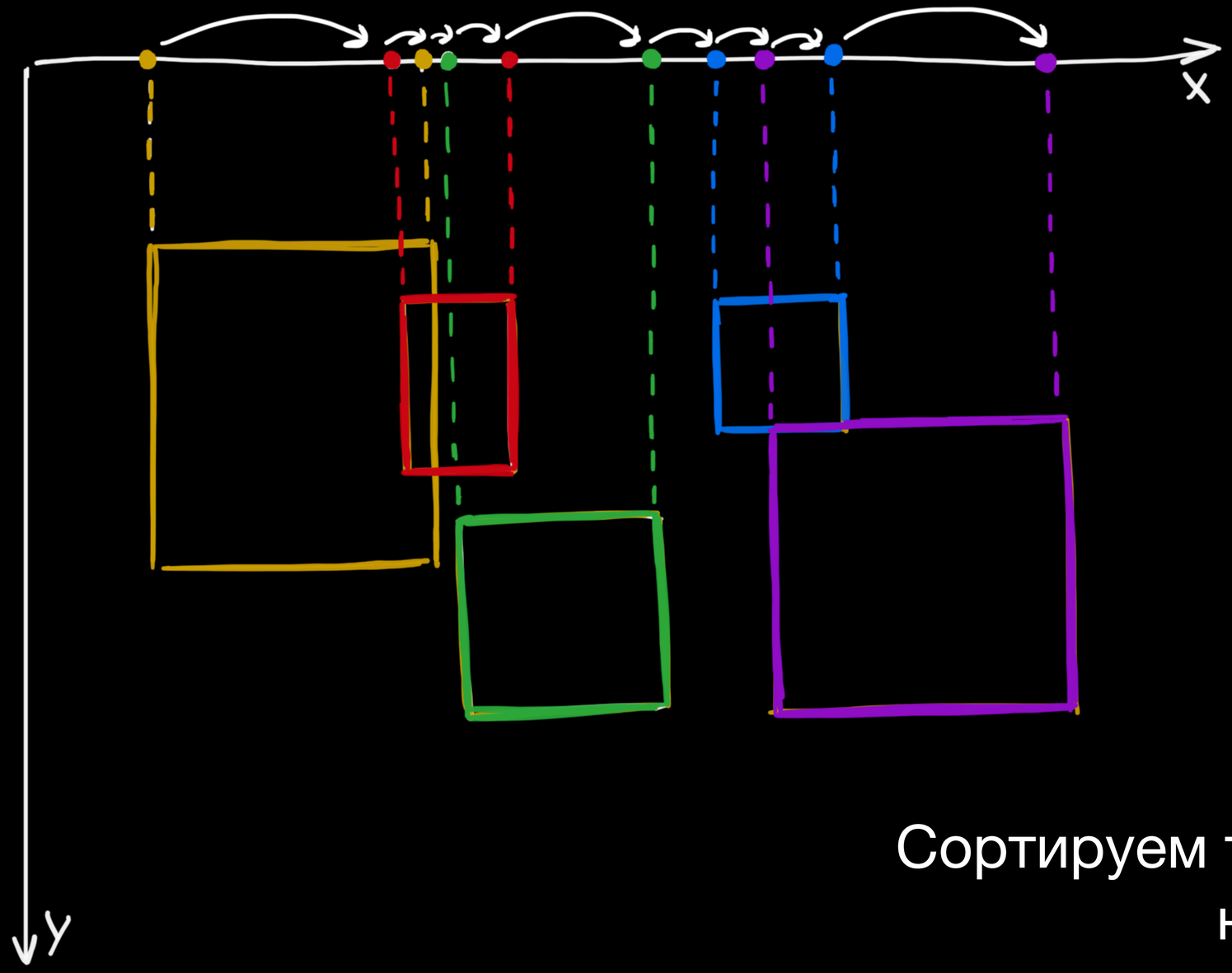
Есть фигуры

Строим ограничивающие объемы

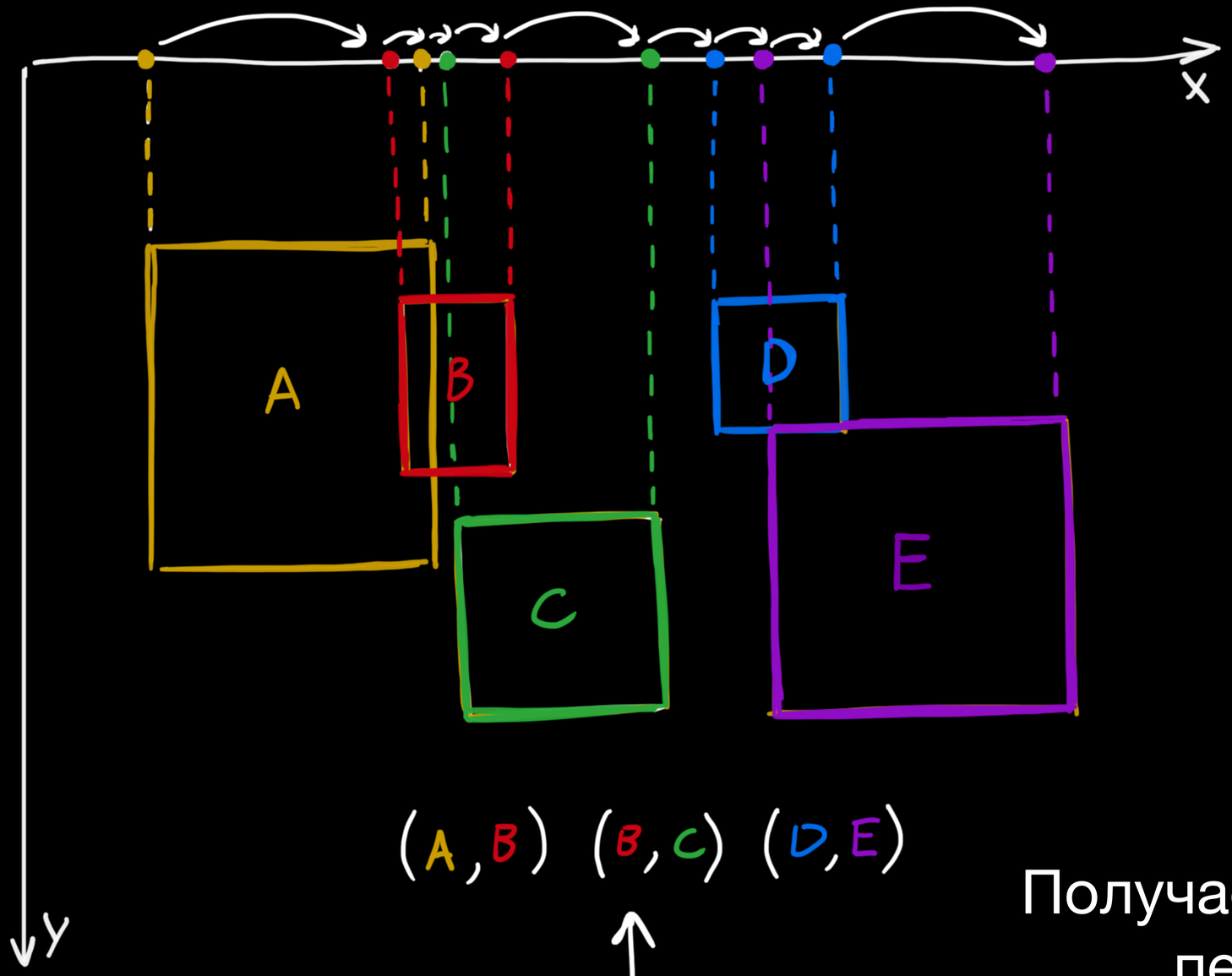




Проецируем на ось

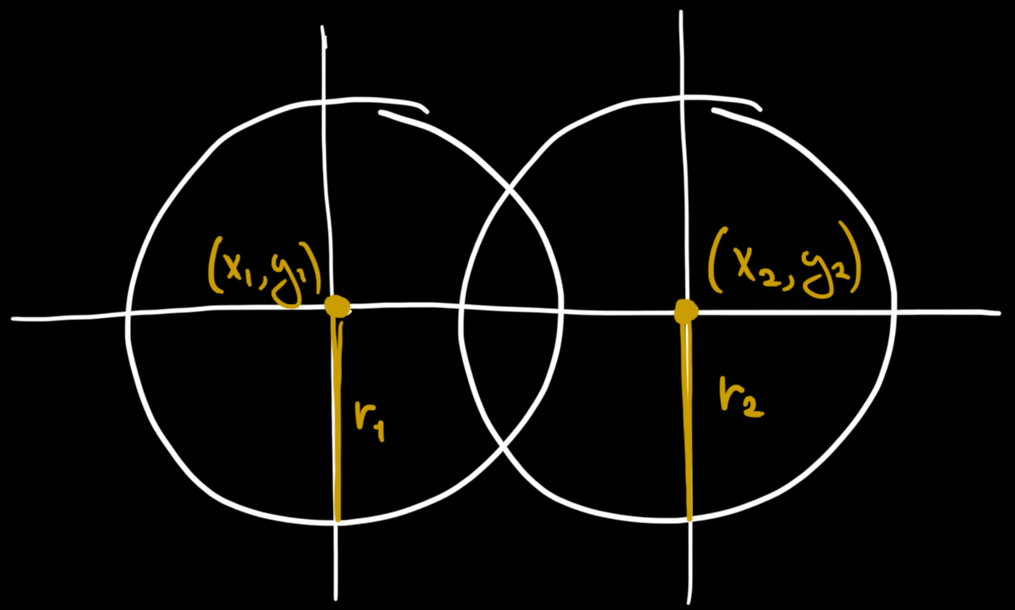


Сортируем точки и проходимся по ним в цикле



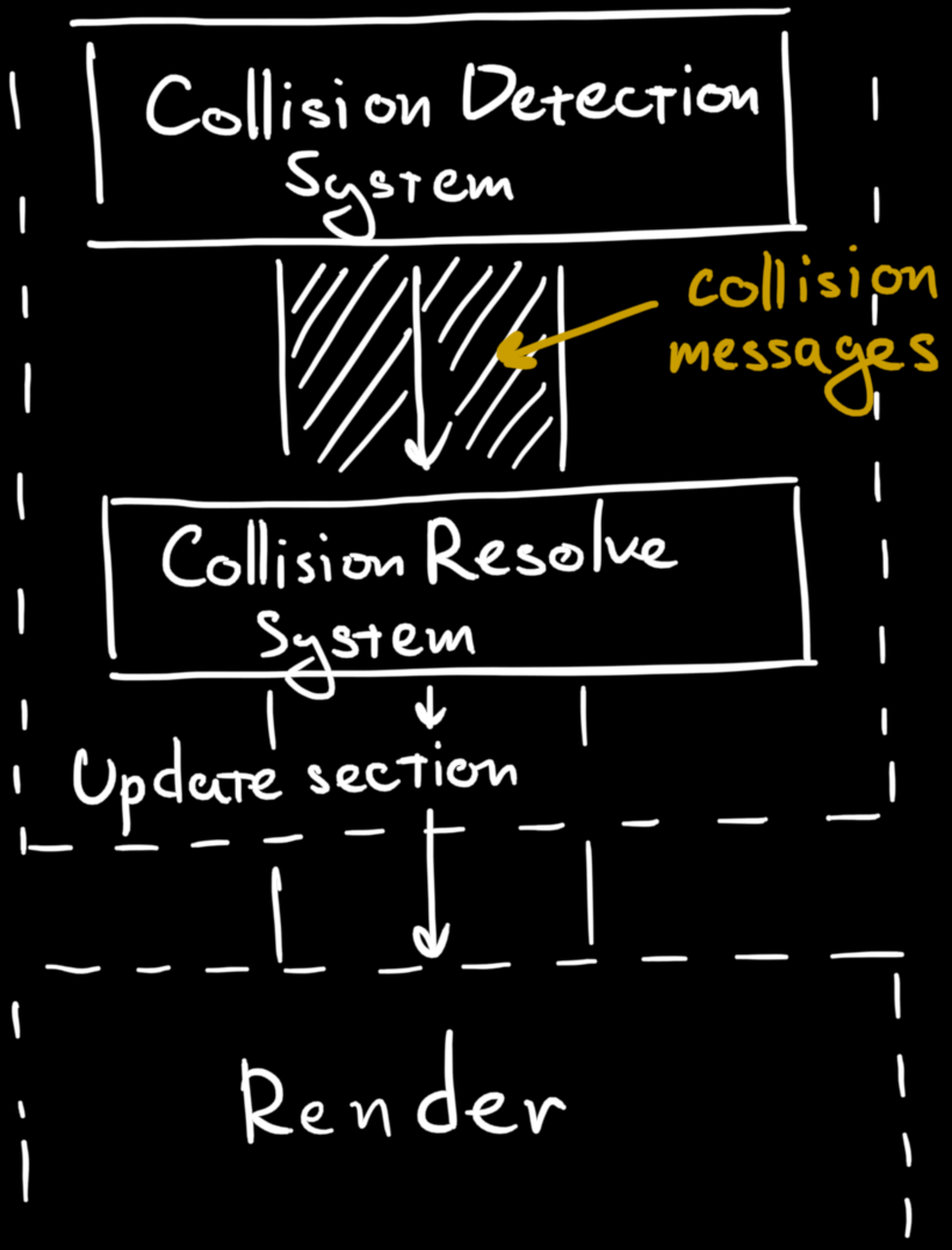
Получаем список потенциально пересекающихся пар

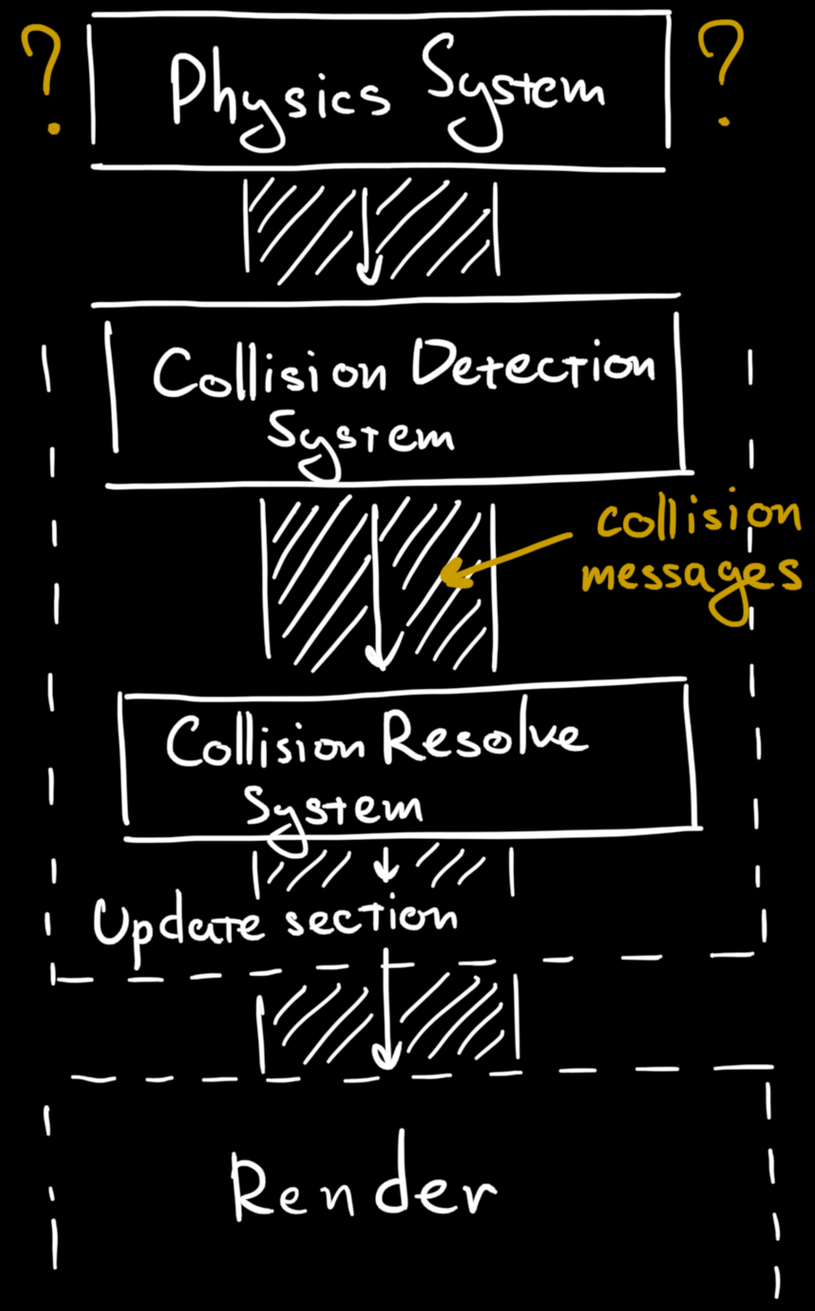
Затем проверяем наверняка



$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < r_1 + r_2$$

```
const message = {
  type: 'COLLISION_ENTER',
  id: gameObject.getId(),
  gameObject: gameObject,
  otherGameObject: otherGameObject,
}
```



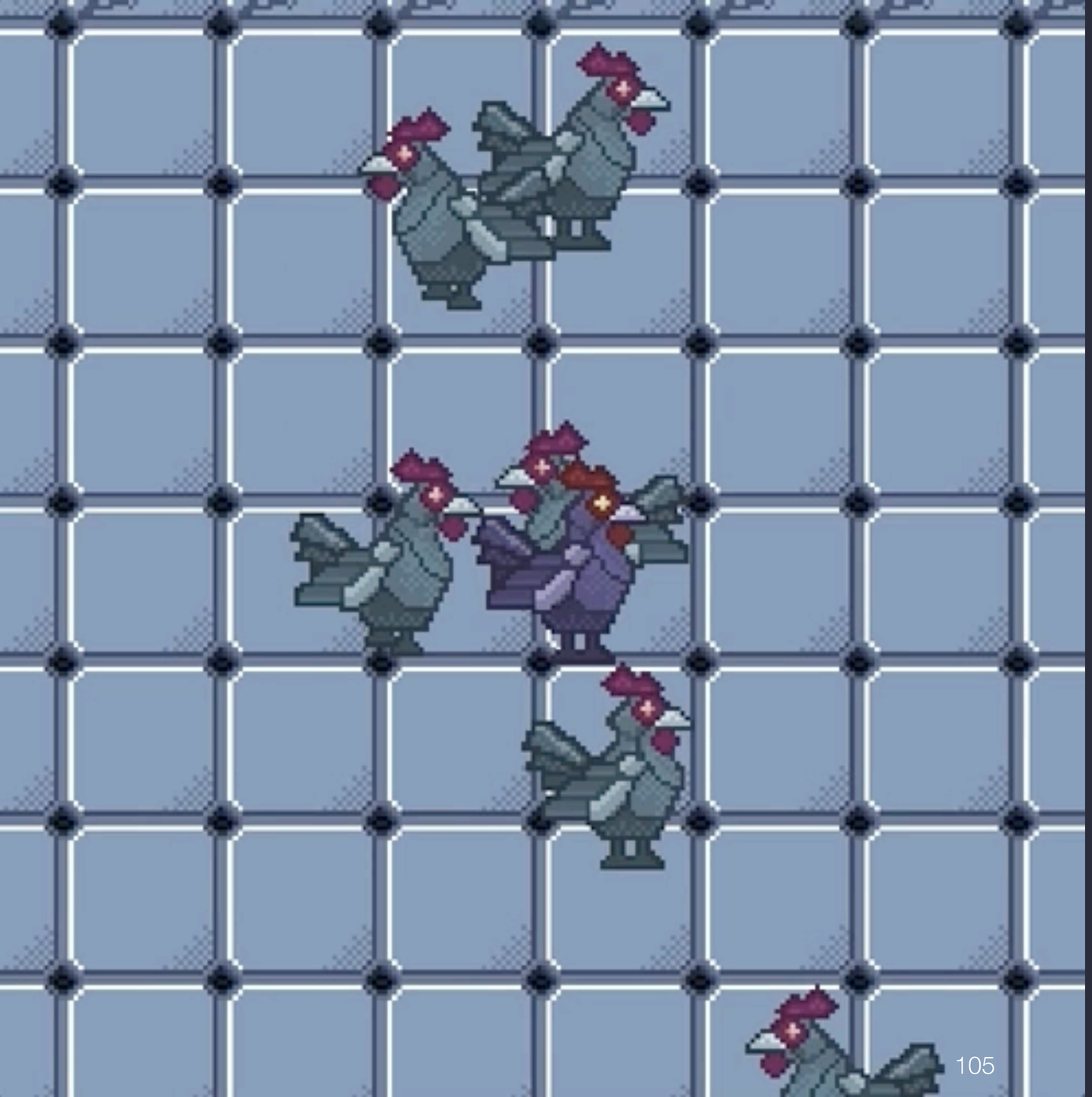



```
class MessageBus {
  constructor() {
    this._messages = {};
  }

  send(message, delay = false) {}

  get(messageType) {}

  delete(messageType) {}

  clear() {}
}
```

```
{
  "name": "rigidBody",
  "config": {
    "mass": 1.75,
    "useGravity": false
  }
},
{
  "name": "colliderContainer",
  "config": {
    "type": "boxCollider",
    "collider": {
      "sizeX": 21,
      "sizeY": 7,
      "centerX": 0,
      "centerY": 17
    }
  }
},
```

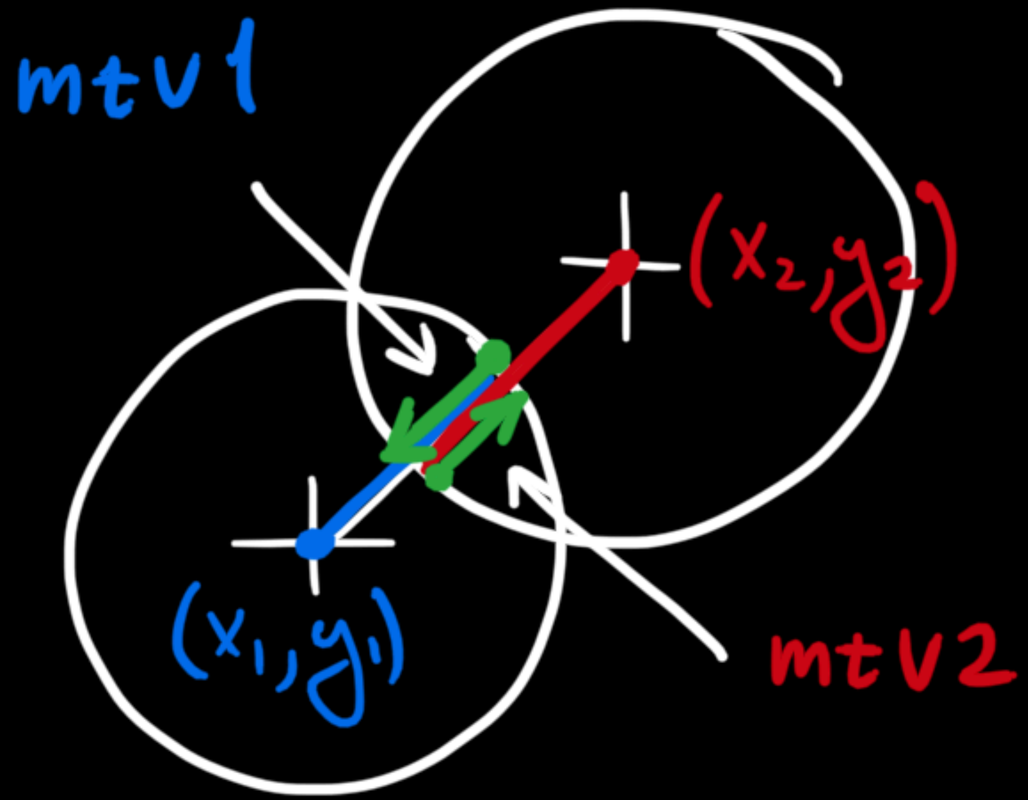
Отмечаем что объект твердый (непроходимый)

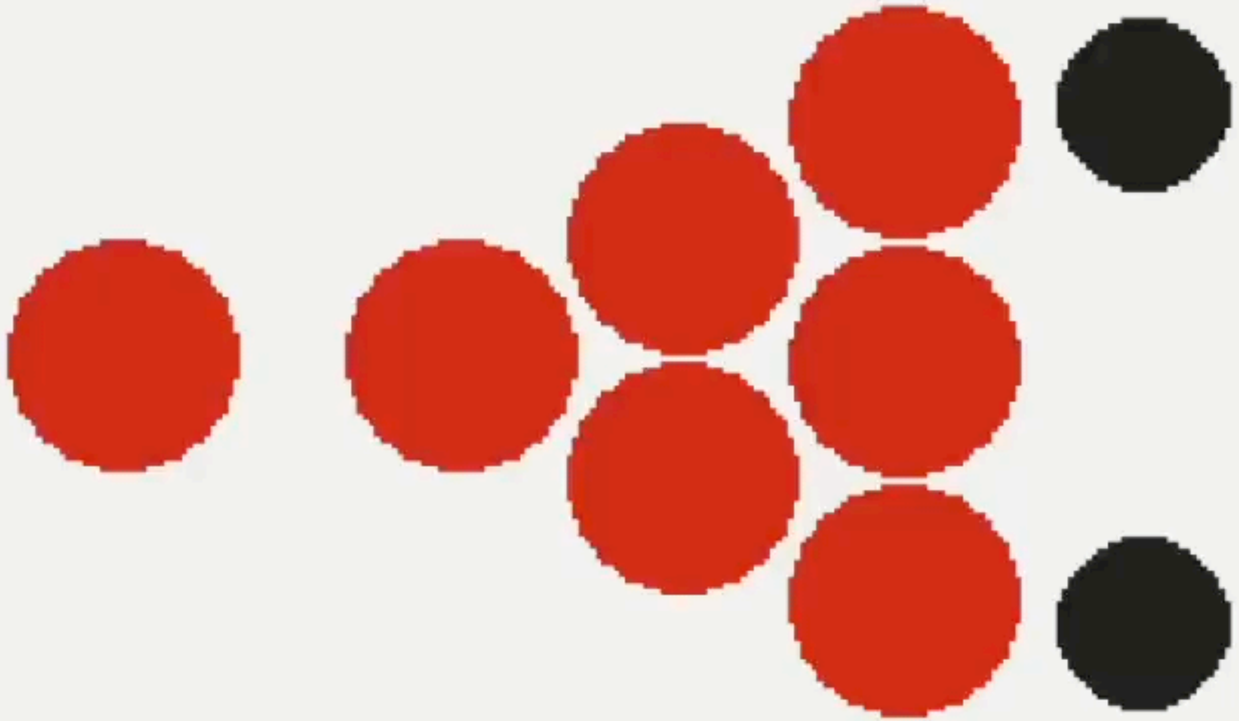
В принципе ок



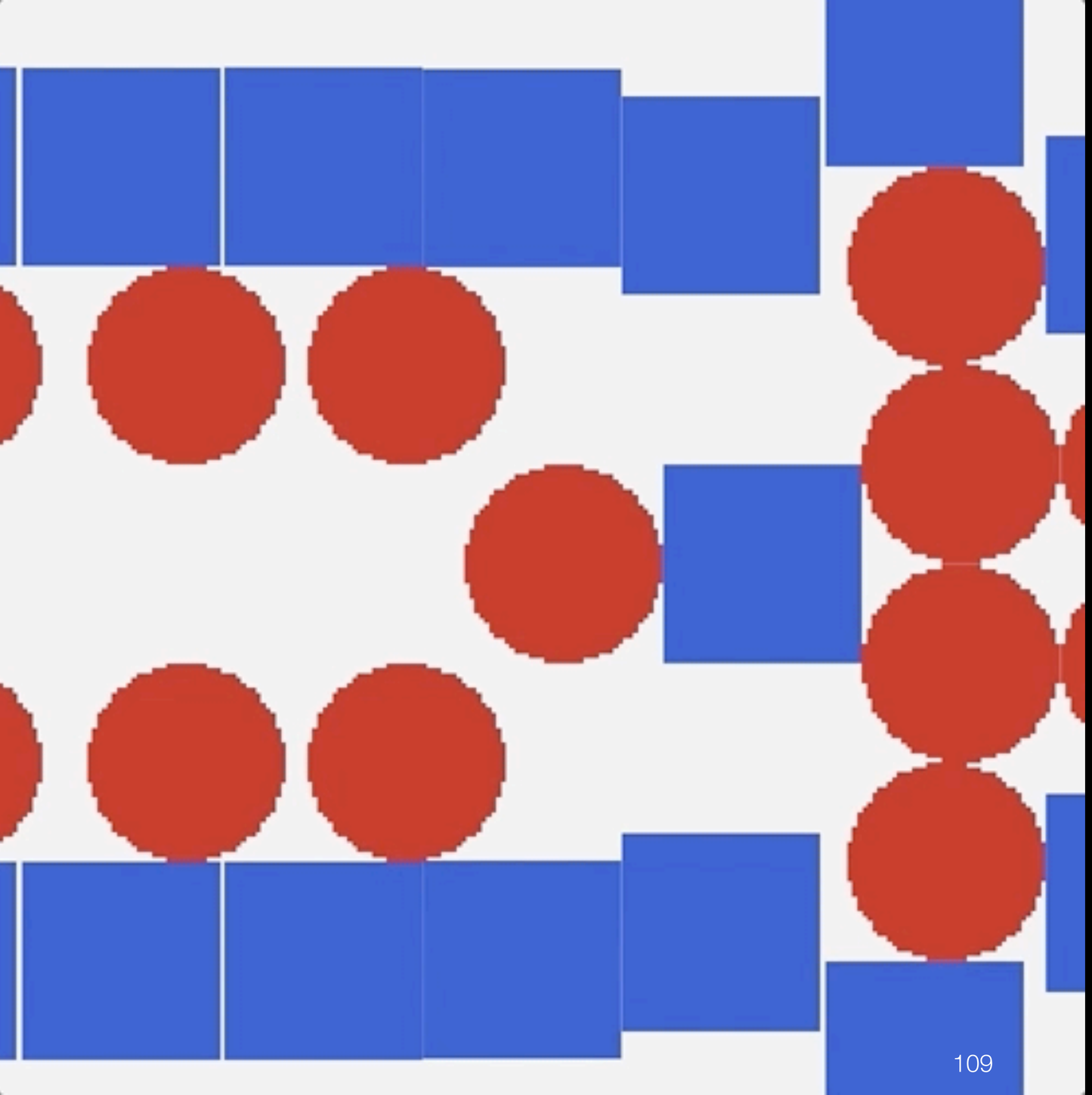
Не ок

```
const message = {
  type: 'COLLISION_ENTER',
  id: gameObject1.getId(),
  gameObject1,
  gameObject2,
  mtv1,
  mtv2,
};
```



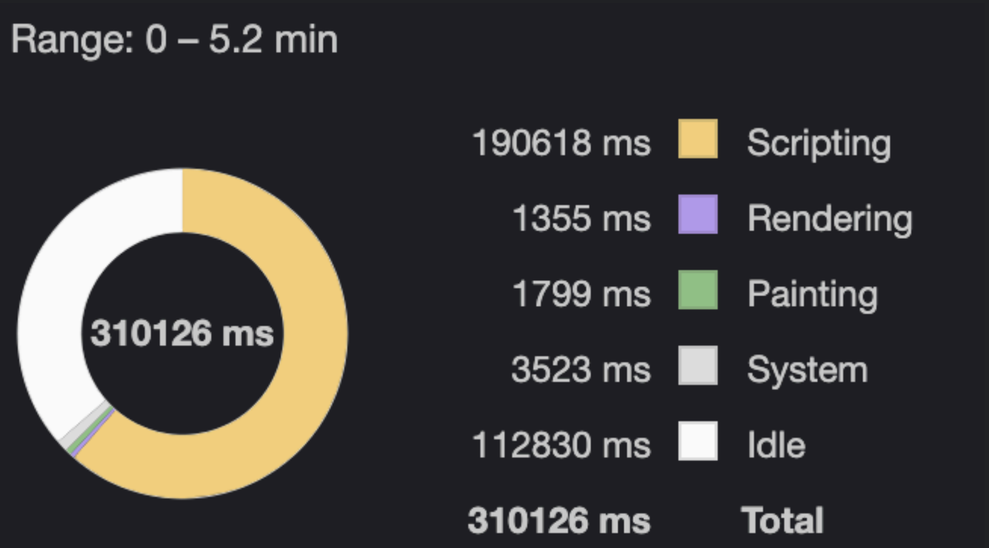


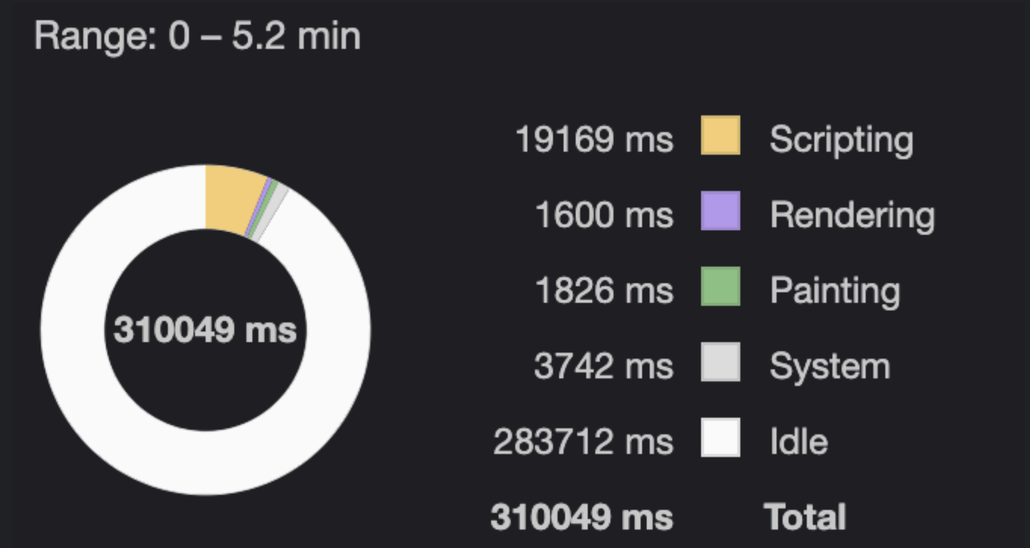
Теперь намного лучше



Но не идеально

Стало ли быстрее с широкой фазой





18694.9 ms 82.7 %	Animation Frame Fired	
17706.3 ms 78.4 %	Function Call	
15532.4 ms 68.7 %	tick	webpack-internal:///61:35
14138.4 ms 62.6 %	_processSection	webpack-internal:///61:21
13872.2 ms 61.4 %	(anonymous)	webpack-internal:///61:27
5600.6 ms 24.8 %	process	webpack-internal:///79:255
2084.0 ms 9.2 %	process	webpack-internal:///76:57
1558.6 ms 6.9 %	process	webpack-internal:///67:29

189625.7 ms 97.9 %	Animation Frame Fired	
188678.6 ms 97.4 %	Function Call	
186786.1 ms 96.4 %	tick	webpack-internal:///61:35
185621.2 ms 95.8 %	_processSection	webpack-internal:///61:21
185374.9 ms 95.7 %	(anonymous)	webpack-internal:///61:27
177565.9 ms 91.6 %	process	webpack-internal:///79:255
2231.3 ms 1.2 %	process	webpack-internal:///76:57
1461.3 ms 0.8 %	process	webpack-internal:///67:29

В ~31 раз

Roosters Fight

PLAY

Попробуем еще раз

Ludum Dare

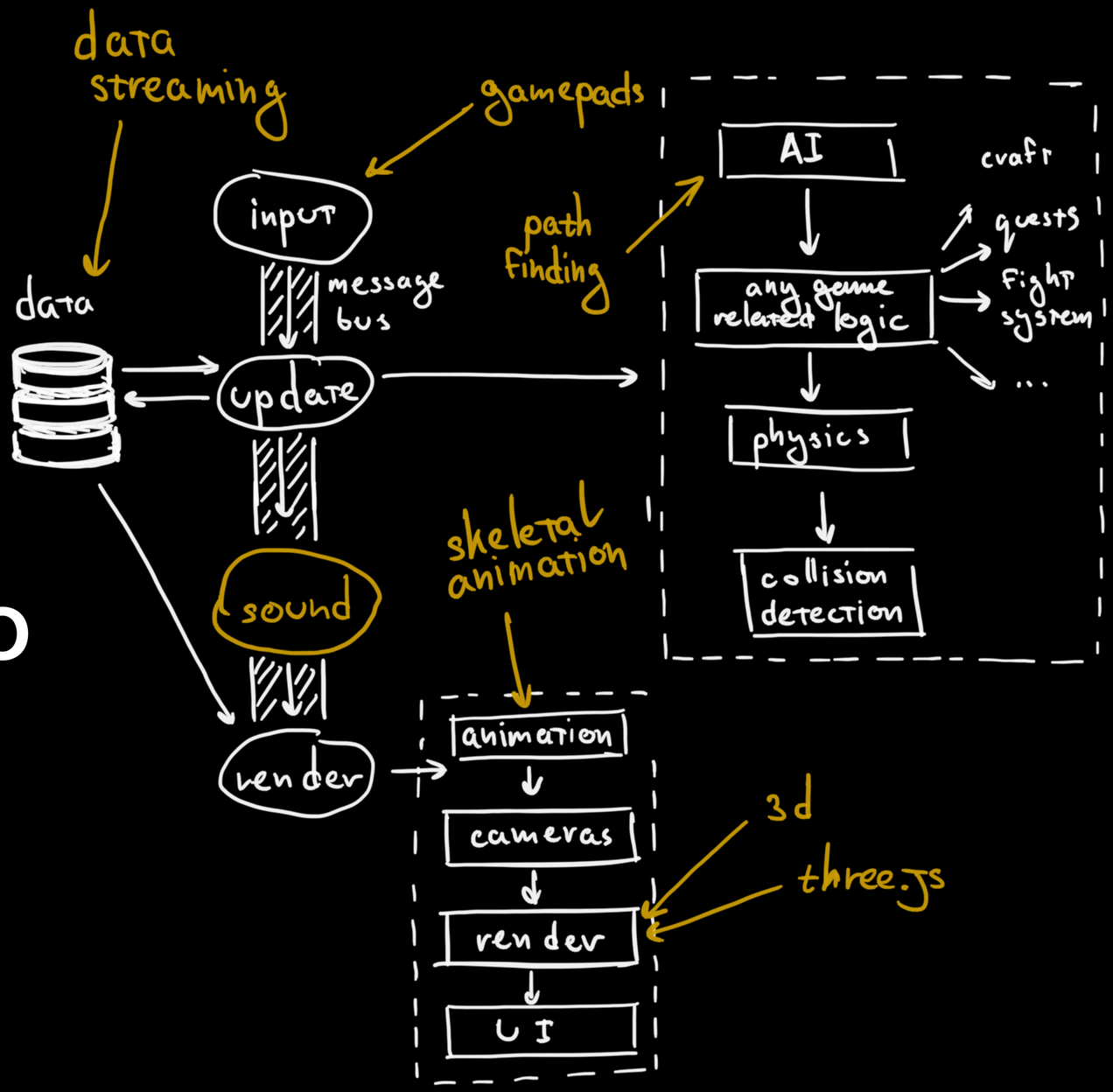
HEALTH: 100

I INVENTORY

Цели достигнуты

- ✓ Разобраться как устроена разработка игр с нуля
- ✓ Создать MVP игрового движка

Продолжать можно бесконечно



npm

Теперь я опенсорсер

На этом все

Движок

Мой блог