lssx, LÖVE Space Shooter X

Artifact

Extended Project Qualification

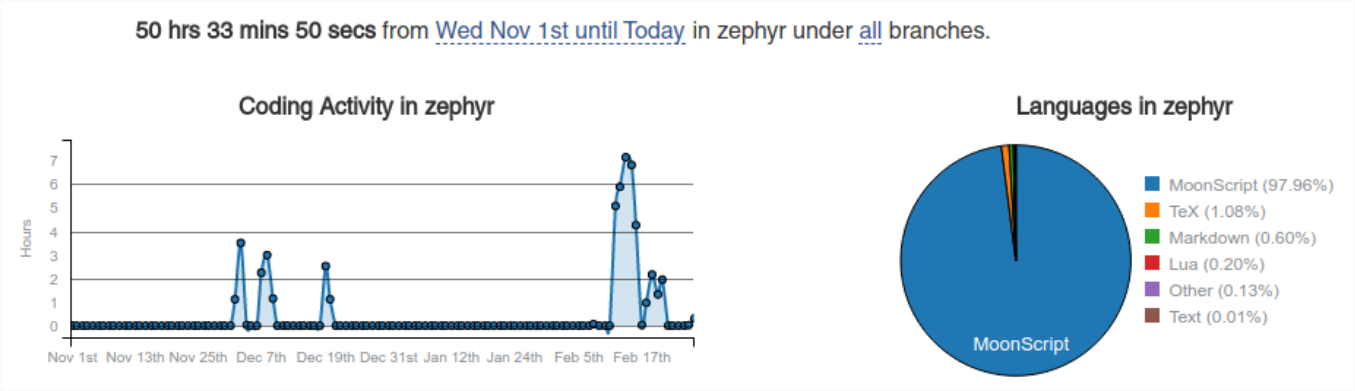
2018

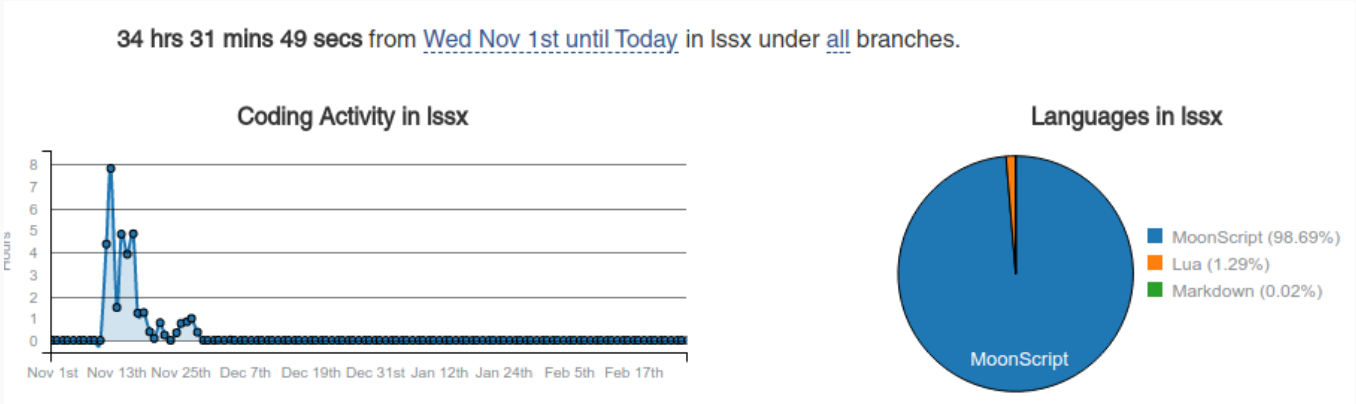
ttxi - JLC

Gameplay and Statistics

`https://youtu.be/LxWgUlpdvgs`

- ~3000 lines of code
- approx. 100 hours spent
- 97 commits





Why lssx?

Several years experience in programming

Never finished a game before

Developing a game requires a multitude of skills ranging from design, mathematics, physics and logic.

Future university degree requires programming.

Fun.

Project aims

Aims were to create a small-scoped space-esque shoot-em-up with semi-realistic physics and retro graphics, akin to **Asteroids**, **Space Invaders**.

- Finish it
- Enjoyable, *re-playable* and simple experience
- Release onto itch.io

Inspiration & Research

Cold war paranoia, fantasy systems (Star Wars ICBM defence) - Training simulator for cold-war pilots.

- BYTEPATH
- Reassembly
- DATA WING

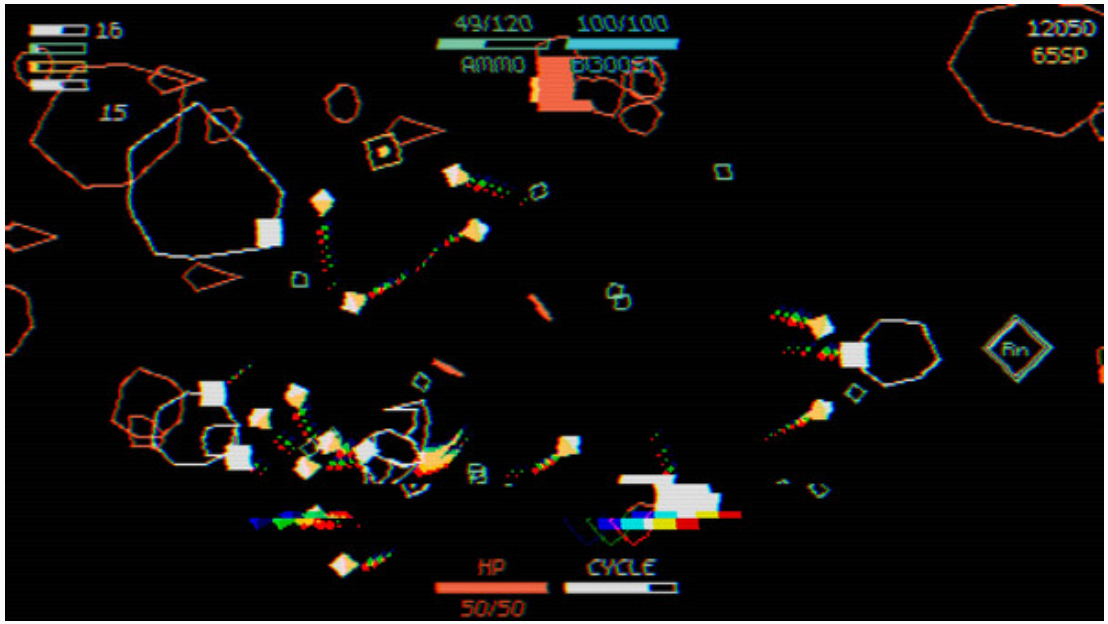


Figure 1: BYTEPATH

Reassembly

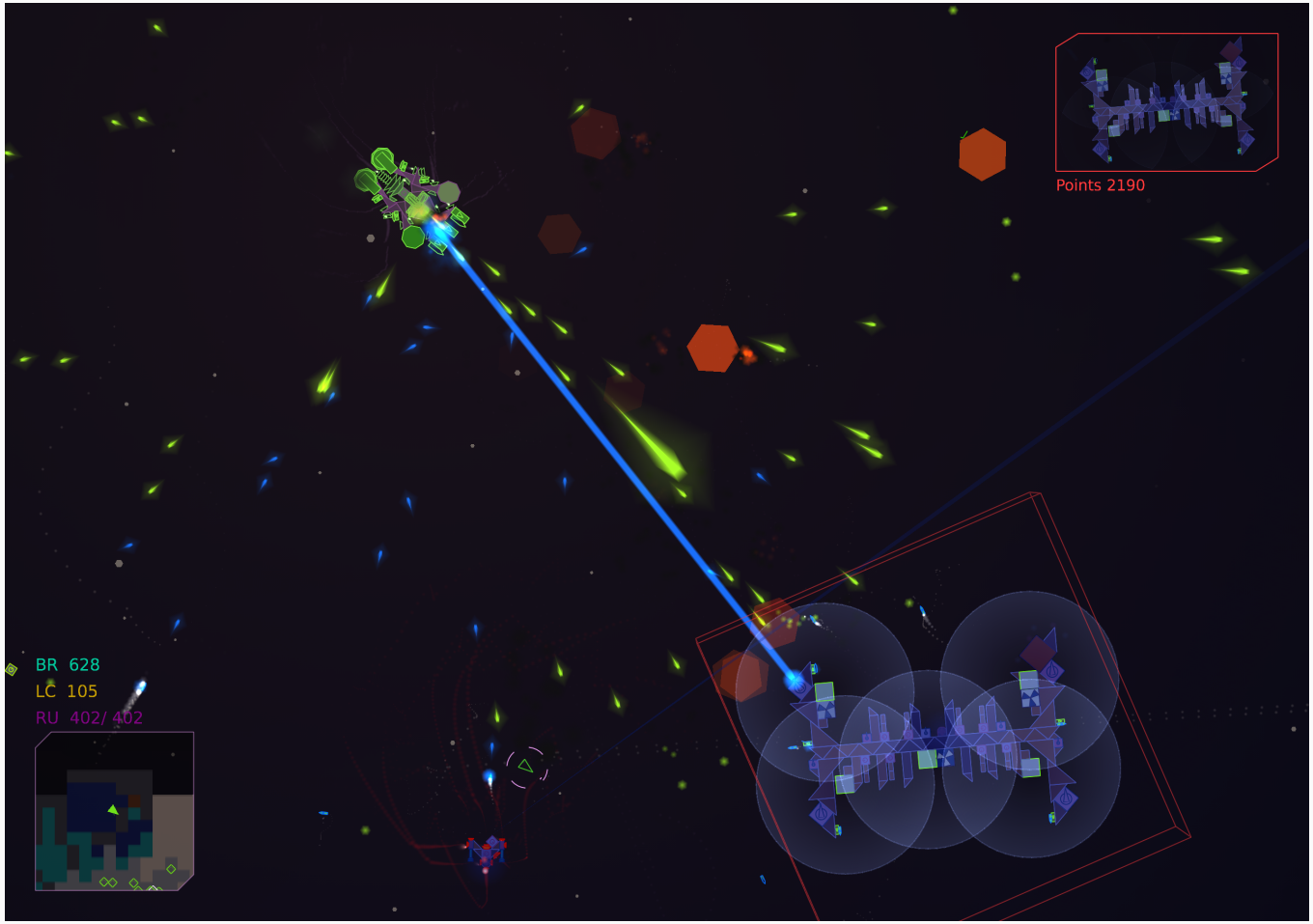


Figure 2: Reassembly

Vector graphics, heavy use of lighting and shaders, minimal UI

DATA WING

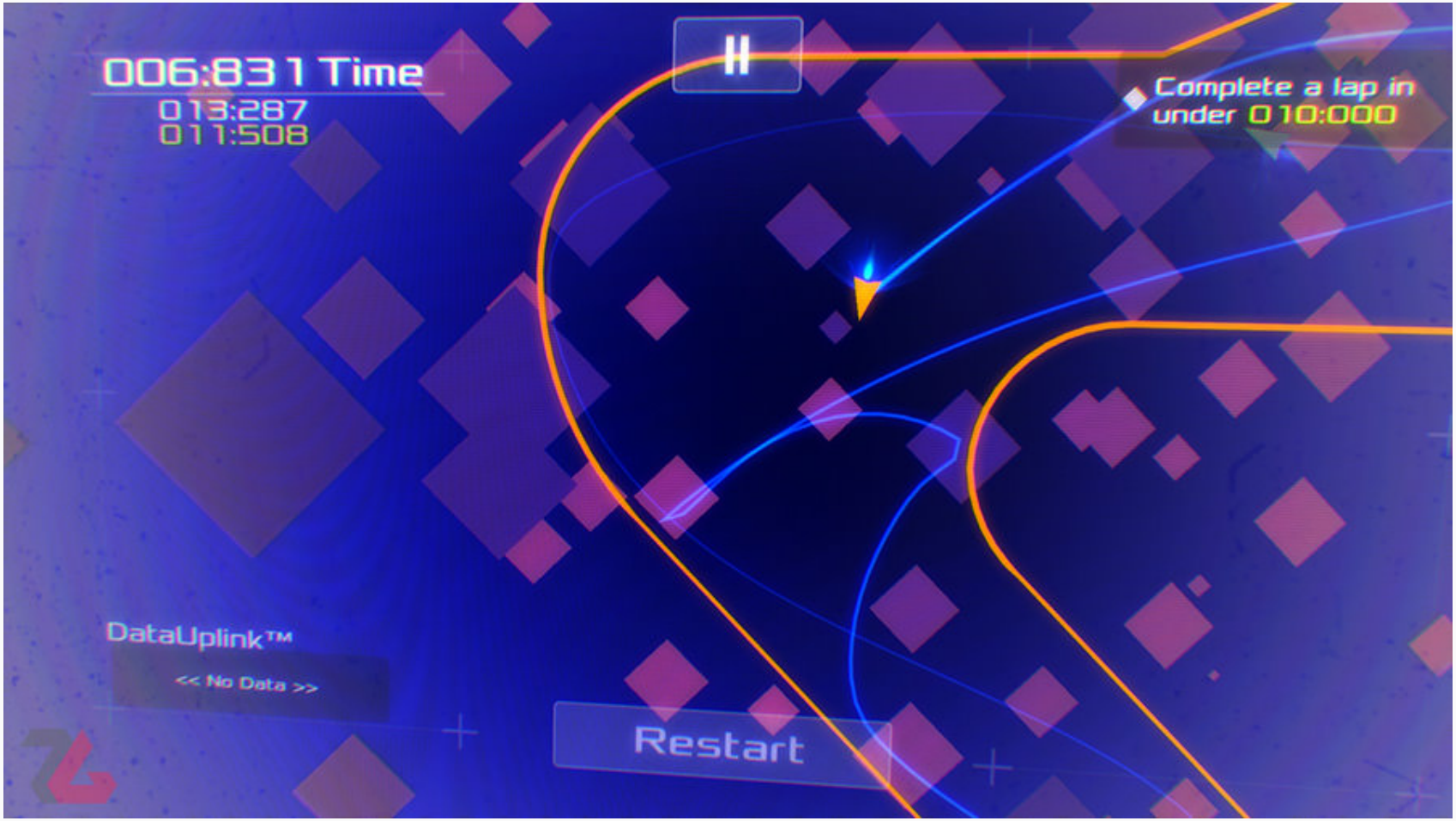


Figure 3: DATA WING

Basic yet enjoyable movement mechanics, camera parallax.

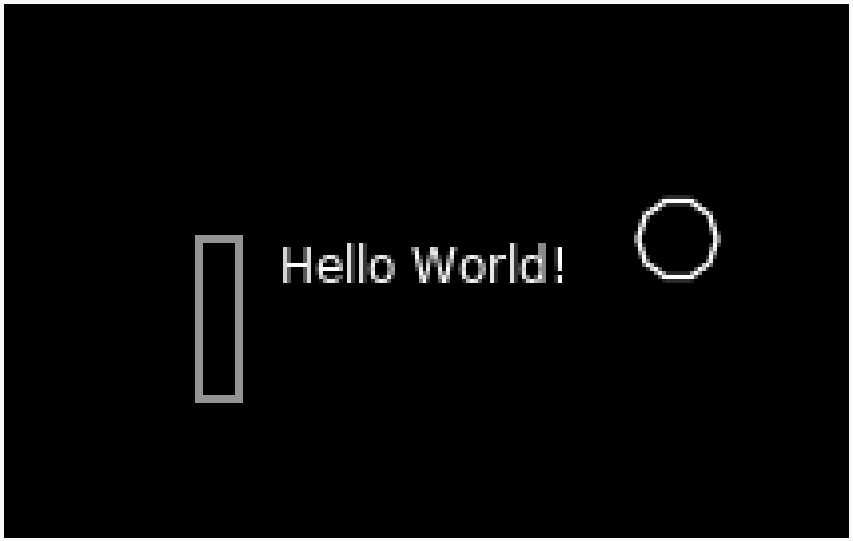
Architecture

- **Lua**, a powerful, efficient, lightweight, embeddable scripting language, used in programs such as Adobe Lightroom and Civ5.
- **LÖVE**, 2D game development framework, used in commercial games such as "*Move or Die*".
- **Box2D**, 2D physics engine used to realistically simulate interaction between rigid bodies, in development for over 10 years.

Decided to use **Lua** and **LÖVE** because of prior experience with both, I didn't want to learn a new language whilst making a new project.

Lua and LÖVE

```lua
function love.draw()
  love.graphics.print("Hello World!", 400, 300)
  love.graphics.circle("line", 500, 300, 10)
  love.graphics.rectangle("line", 380, 300, 10, 40)
end
```



Box2D

Box2D powers almost all physics interactions within the game, it's proven to be reliable and fast and also has plenty documentation to learn from.

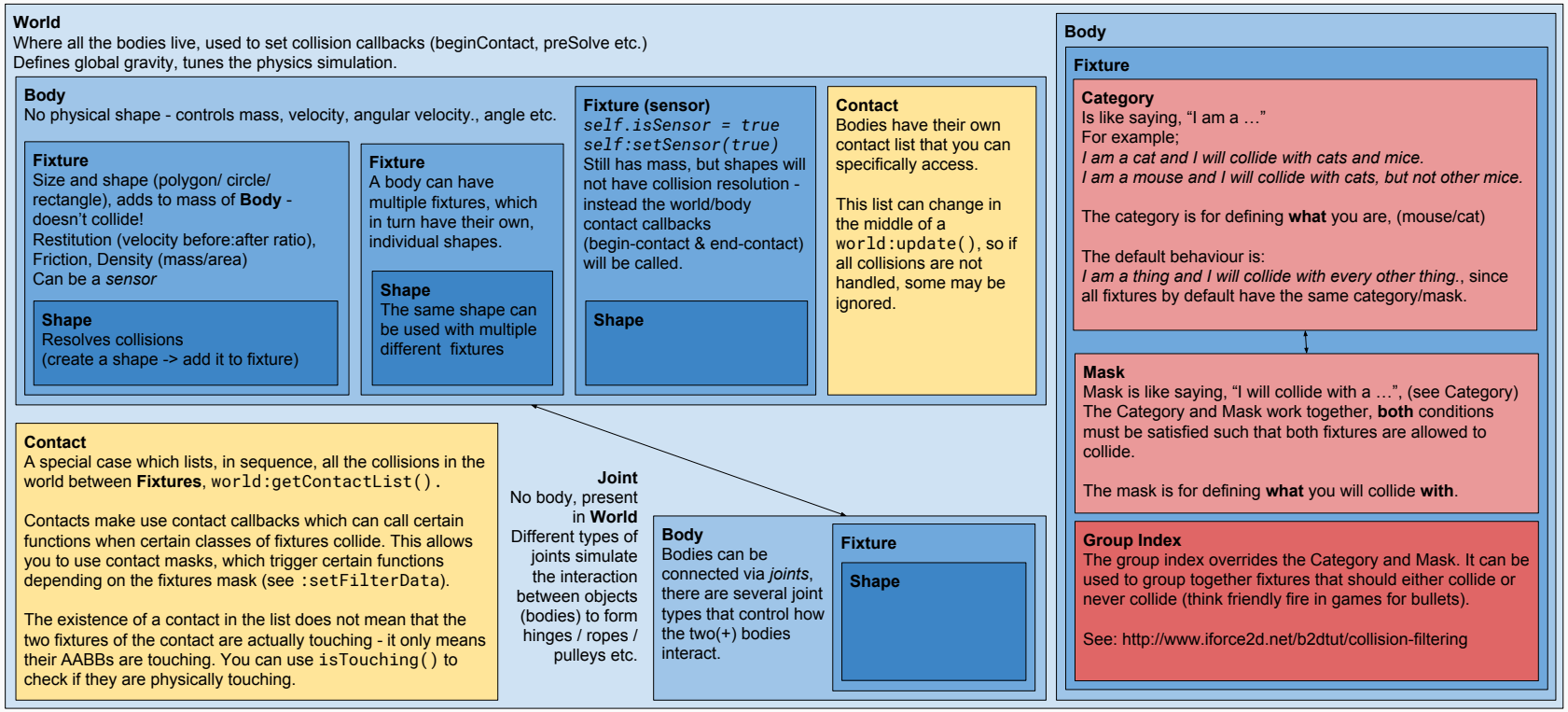


Figure 4: Anatomy of Box2D

Important Box2D concepts

- **Body**, defines properties of an object you cannot see, density, location, rotational inertia and others.
- **Fixture**, used to define material properties of an object, e.g. friction, restitution.
- **Shape**, defines the actual physical shape for collisions.

Multiple **fixtures** can be added to a **Body** to create different forms.

Shapes inside **Fixtures**, **Fixtures** inside **Bodies**.

How do you even write a game?

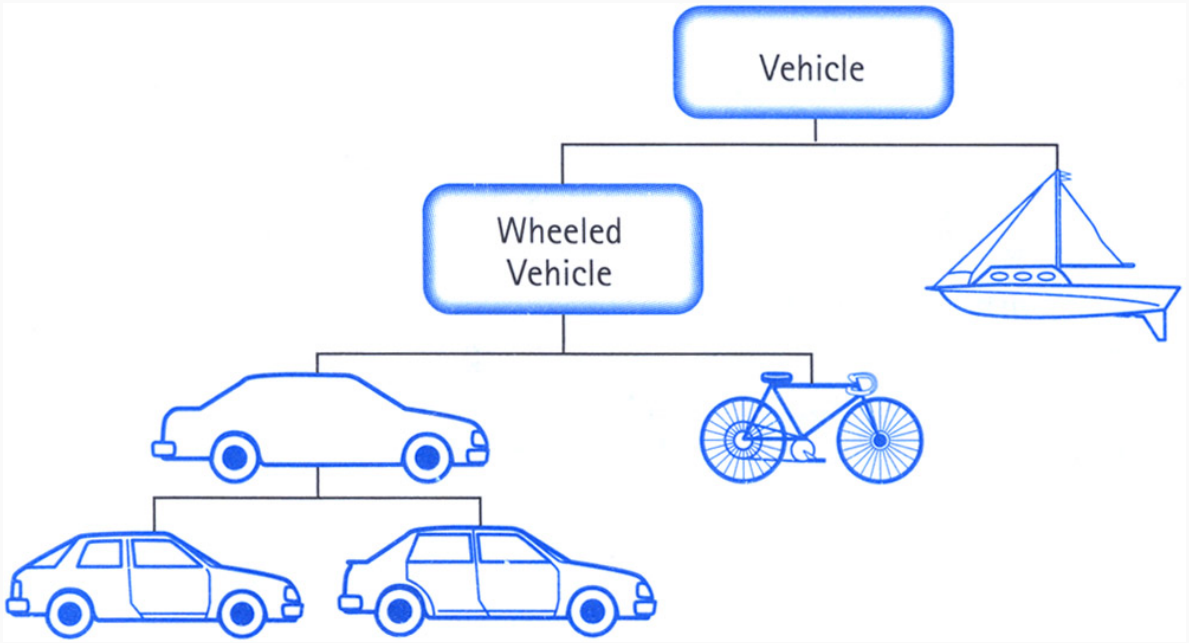
Idea $\rightarrow$ Program idea $\rightarrow$ Test $\rightarrow$ Debug / improve $\rightarrow$ Repeat

Organization is **paramount**, but not *that* much.

Most popular ways of developing bigger-than-small projects is either with an **Entity Component System** or via **Object Oriented Programming**.

Object Oriented Programming

Object Oriented Programming is a paradigm which attempts to define the behaviour of real world objects via inheriting behaviours.

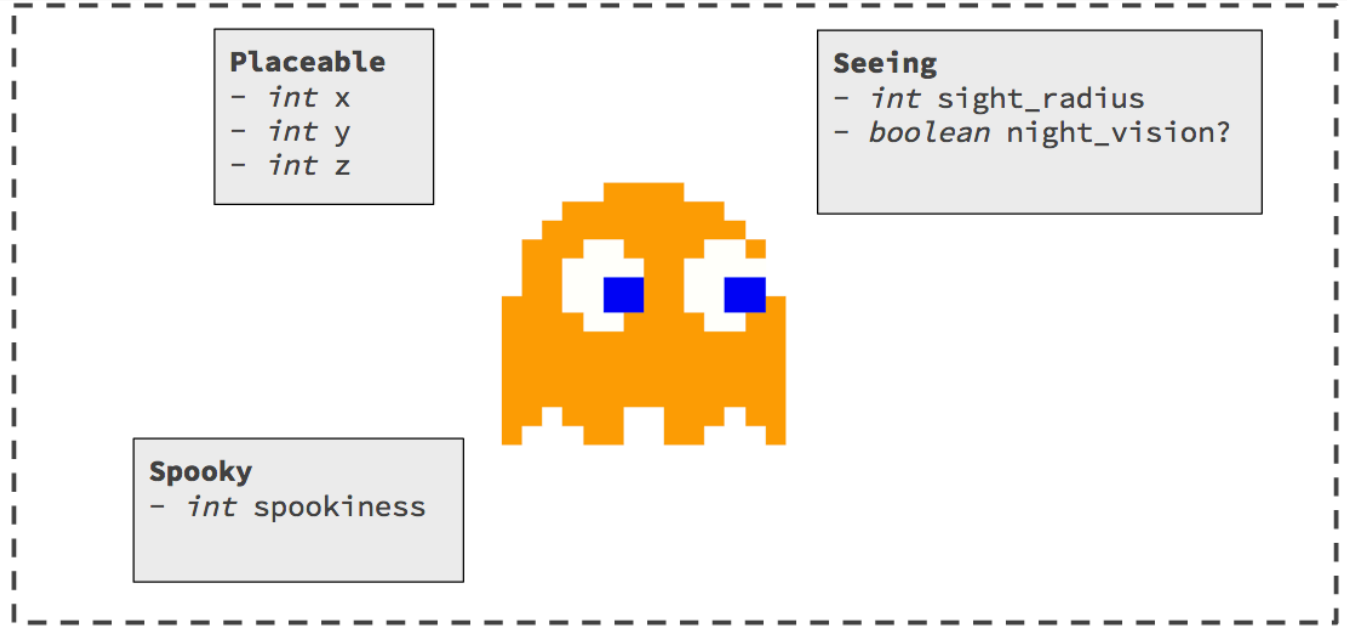


OOP allows for Polymorphism, Encapsulation and Abstraction. Large projects can be created in a sane, organised fashion.

Entity Component System

ECS follows composition over inheritance.
Every entity consists of components which add or define additional behaviour. ECS is better for very large projects because of it's inherent modularity.

A Pacman ghost has some of the following behaviours,



ECS or OOP?

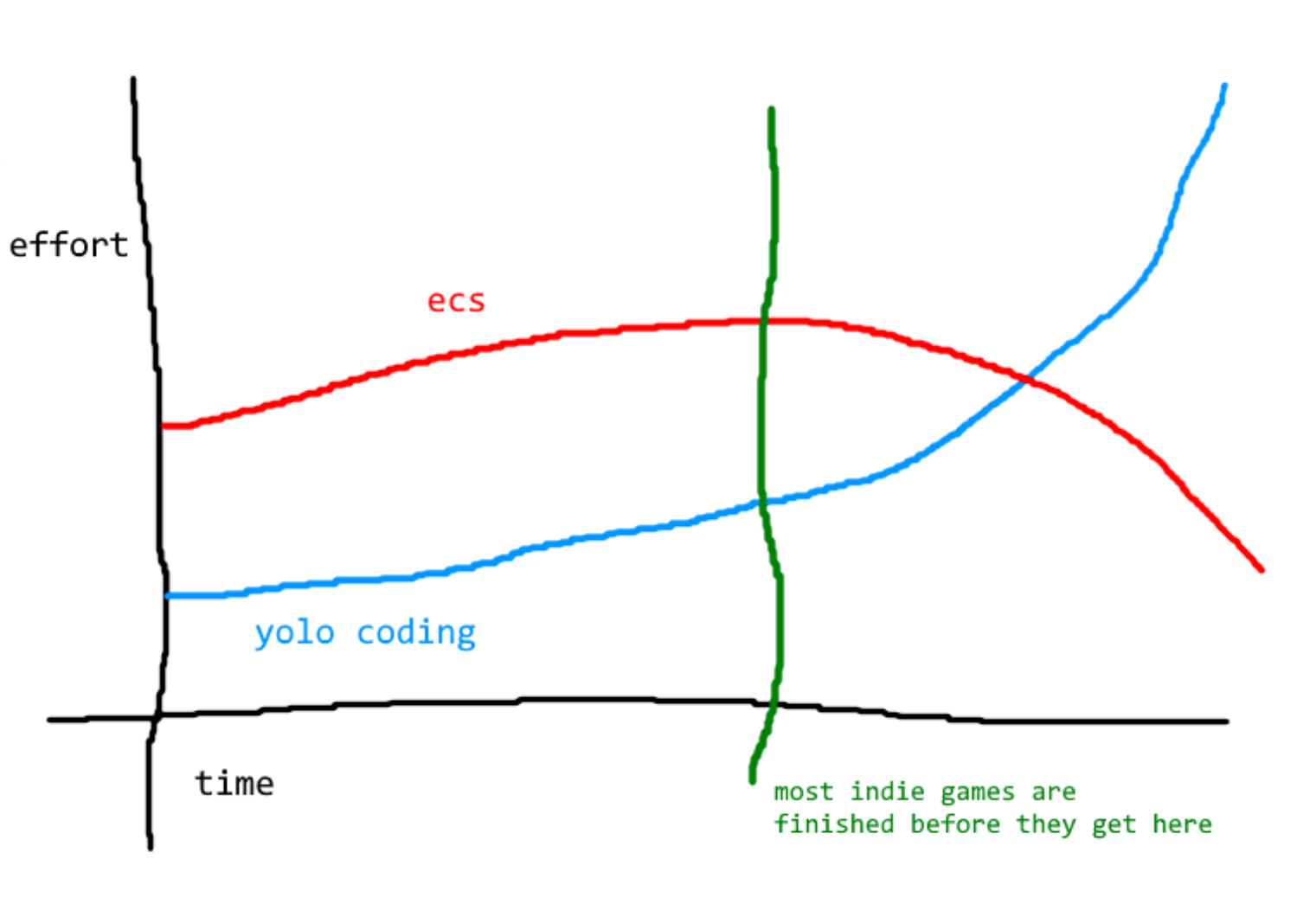


Figure 5: https://github.com/SSYGEN/blog/issues/24

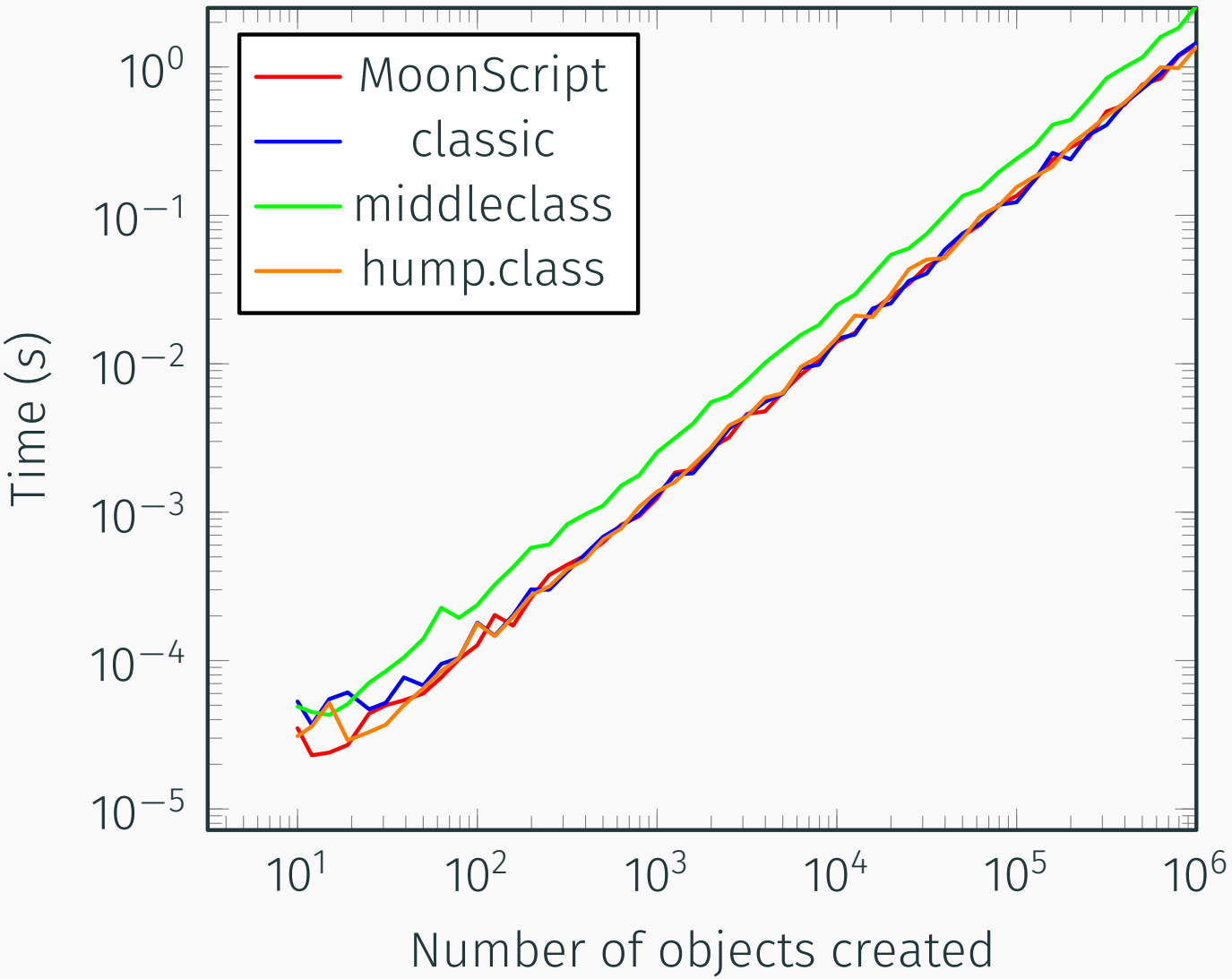
Choosing an OO library

A **library** is a set of reusable functions that perform a set of usually complex tasks that would take a long time to develop yourself.
Lua doesn't natively support *OO*, however the Lua community have created a number of libraries, that allow you to do OOP with Lua.
A series of tests was performed on the most popular OO libraries to see which was the fastest/memory efficient.

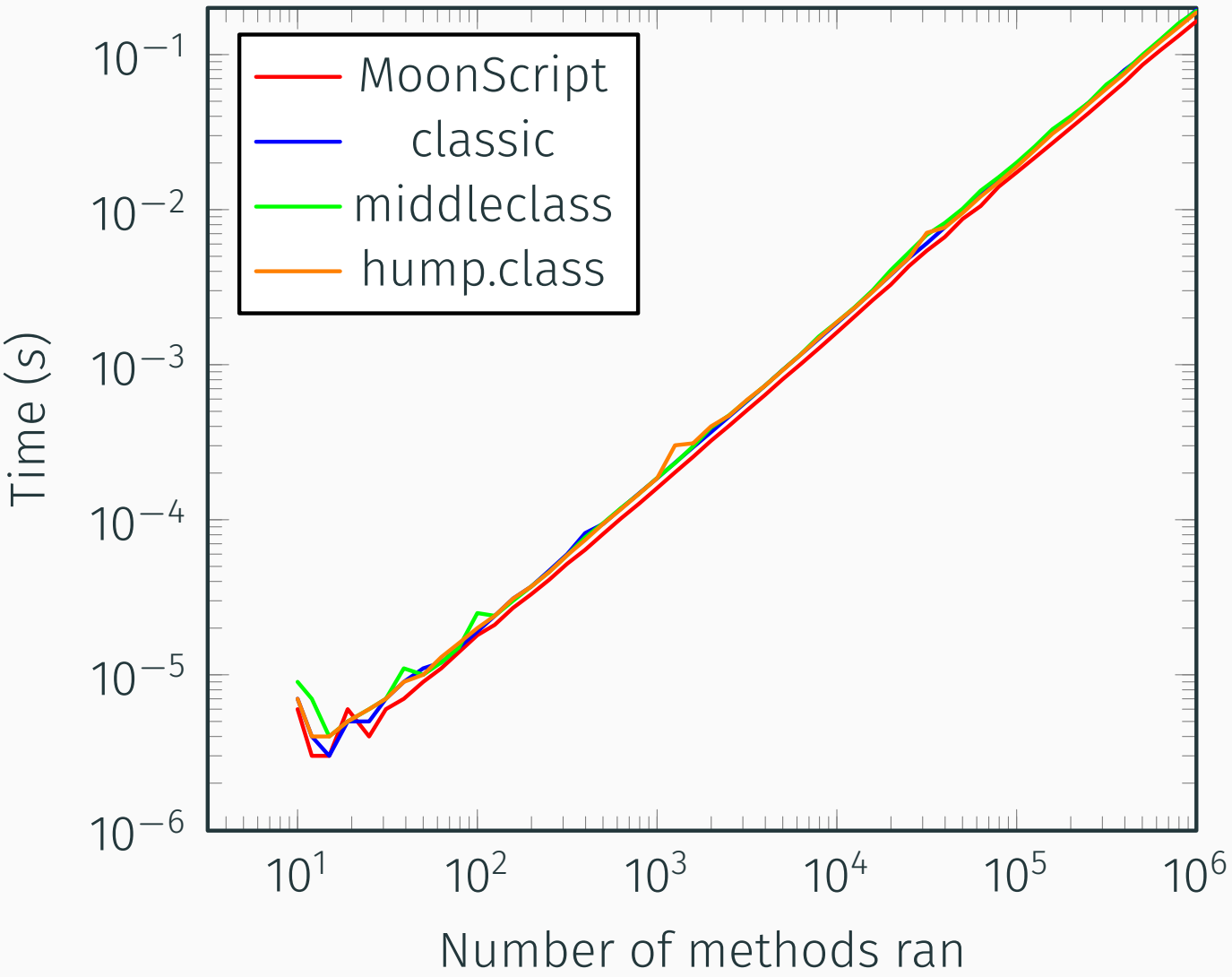
- Creating instances of objects
- Performing methods
- Testing inheritance

From 10 to 1 million objects.

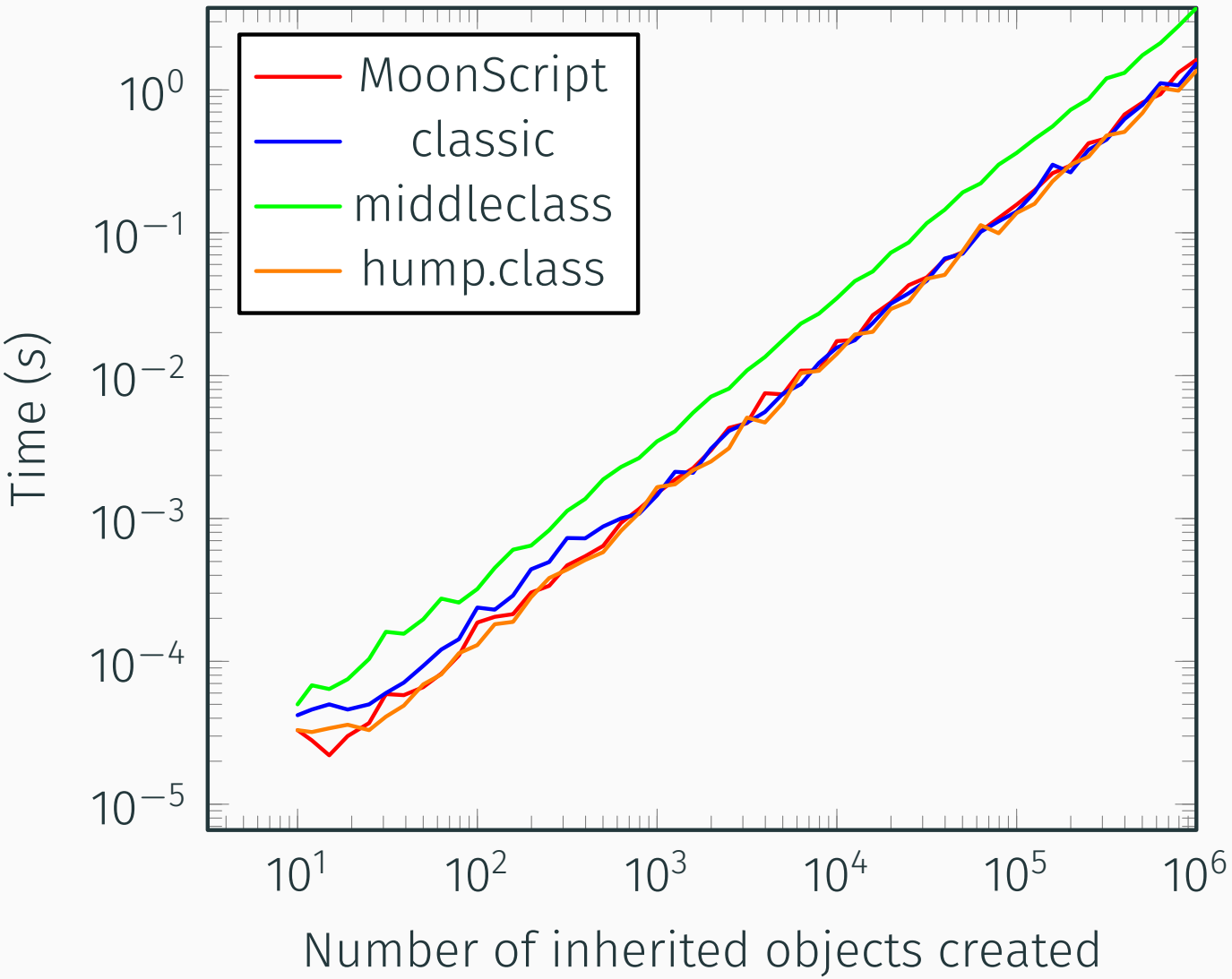
Creating objects



Performing methods



Testing inheritance



Results and conclusion

MoonScript is the leader when dealing with a smaller amount of objects (< 100).

MoonScript is not a library, but is a dynamic scripting language that compiles into Lua, so it can be used with LÖVE.

MoonScript chosen as it has the fastest OO, has greater readability because of the reduced syntactic sugar and therefore errors less likely to be made.

$$\text{MoonScript} \rightarrow \text{Lua} \Rightarrow \text{LOVE + Box2D}$$

MoonScript compiled into Lua

The following MoonScript code:

```
Director.gameStart = () ->
  Timer.every 2, ->
    Pickup(math.random(2000), math.random(2000))
    Asteroid(100+math.random(1800),
      ↪  100+math.random(1800))
```

Is compiled into the following Lua code:

```
Director.gameStart = function()
  return Timer.every(2, function()
    Pickup(math.random(2000), math.random(2000))
    return Asteroid(100 + math.random(1800), 100 +
      ↪  math.random(1800))
  end)
end
```

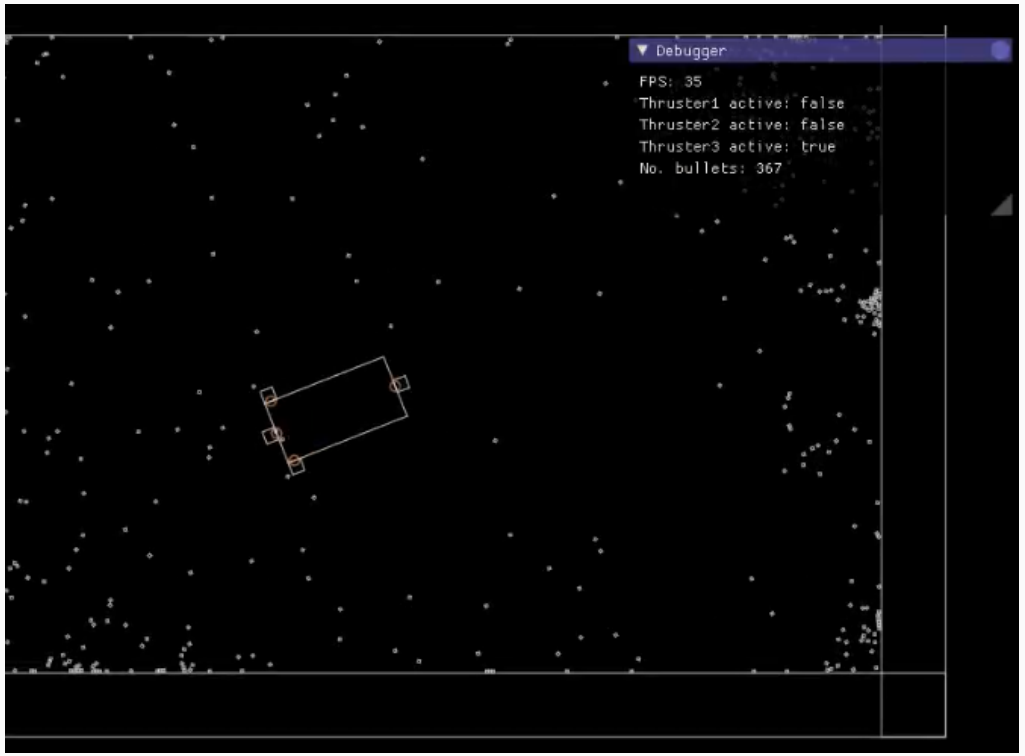
Visually less lines of code.

Programming Philosophies

- **Ease-of-use**, complexity should be avoided, even at the cost of speed
- **Modularity**, the engine should be easily extendable through modular programming
- **Readability**, the code should be easy to read, with most contents' operation being understandable at first-viewing

Rapid prototyping

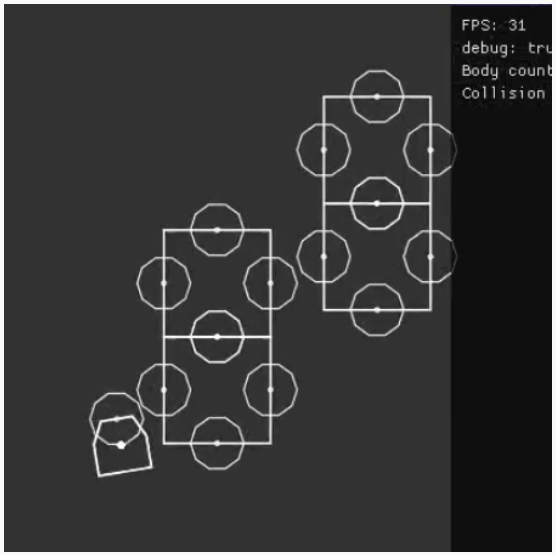
A small prototype was created in under a week to test if the project was feasible.

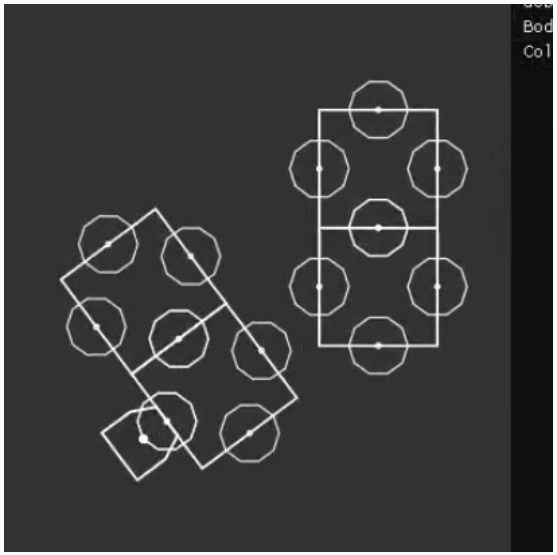


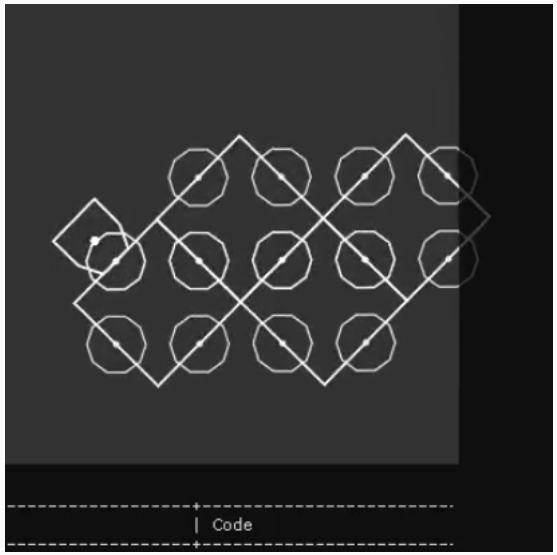
Being a prototype, all the code used had to be re-written to make sure the engine was scalable to the project demands

Deciding on a game loop

Initially toyed around with the idea of destructible ships, thrusters, weapons etc. could be shot off, impairing your ship's abilities - a la **Reassembly**.







After two days of experimenting with the attachment code using a flood fill algorithm I decided this direction would be too complicated.

zephyr

Mid-way through the project I realised the current method of handling collisions was un-scalable and had to be re-worked.

I created **zephyr**, a Box2D wrapper designed to simplify physics management by streamlining collision detection and resolution between Box2D objects.

Approx 750 lines of code.

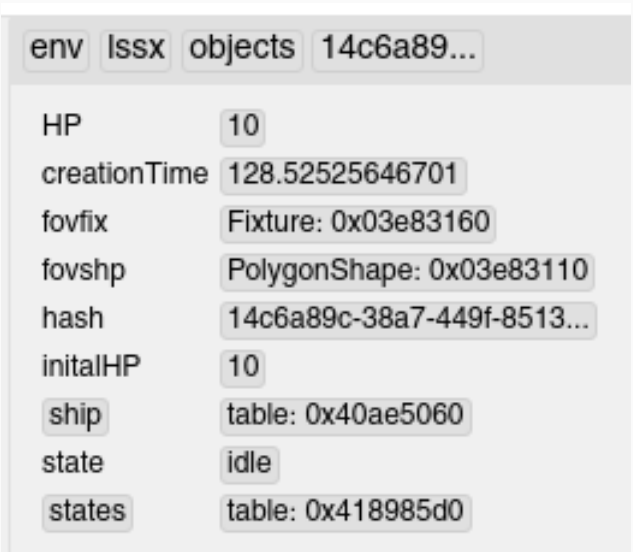
```
Physics.beginContact = (a, b, coll) ->
  -- pass a->b and b->a
  lssx.objects[a\getUserData().hash]\beginContact(b)
  lssx.objects[b\getUserData().hash]\beginContact(a)
```

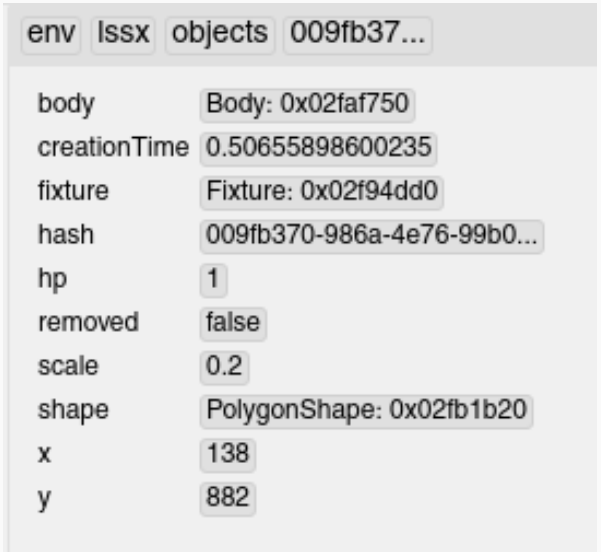
zephyr and Entity Management

The main feature of *zephyr* is it's fast object identification with the use of **Universally Unique IDentifier**'s organised in a hash-table.

A typical **UUID** looks like:
`0264d794-e06a-4a8c-b018-d61aee5aa2b3`

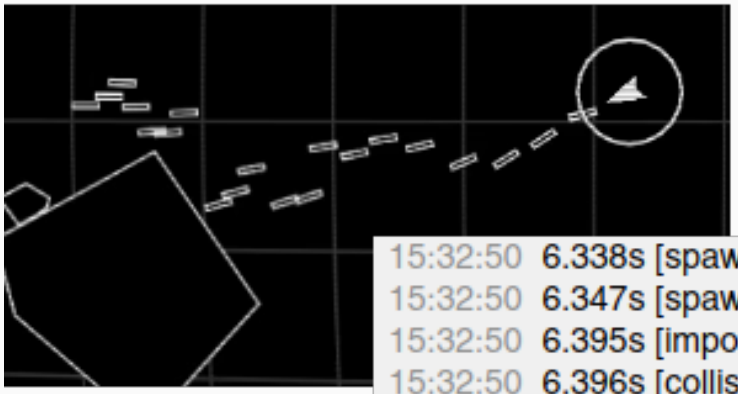
zephyr also allows colliding fixtures to communicate with each other and change each others values by the use of a buffer.





zephyr and Entity Management cont.

A typical interaction between two objects prints the following to the debug log.



```
15:32:50  6.338s [spawn ] Spawned Bullet
15:32:50  6.347s [spawn ] Spawned Bullet
15:32:50  6.395s [important ] beginContact() triggered
15:32:50  6.396s [collision ] -> Bullet, k: 39b9afaf-d39f-4e5f-a46f-31408079c768
15:32:50  6.398s [collision ] -> Asteroid, k: 07db2a45-a8f0-4d93-abe7-411ef3b0a90f
```

This shows a collision between an *Asteroid* and *Bullet*, with their **UUID**'s defined as **k**, each object was found within 2 milliseconds.

Procedural Generation

Procedural Generation is a method of computationally generating content. `https://youtu.be/O9KZFE1G6b0`

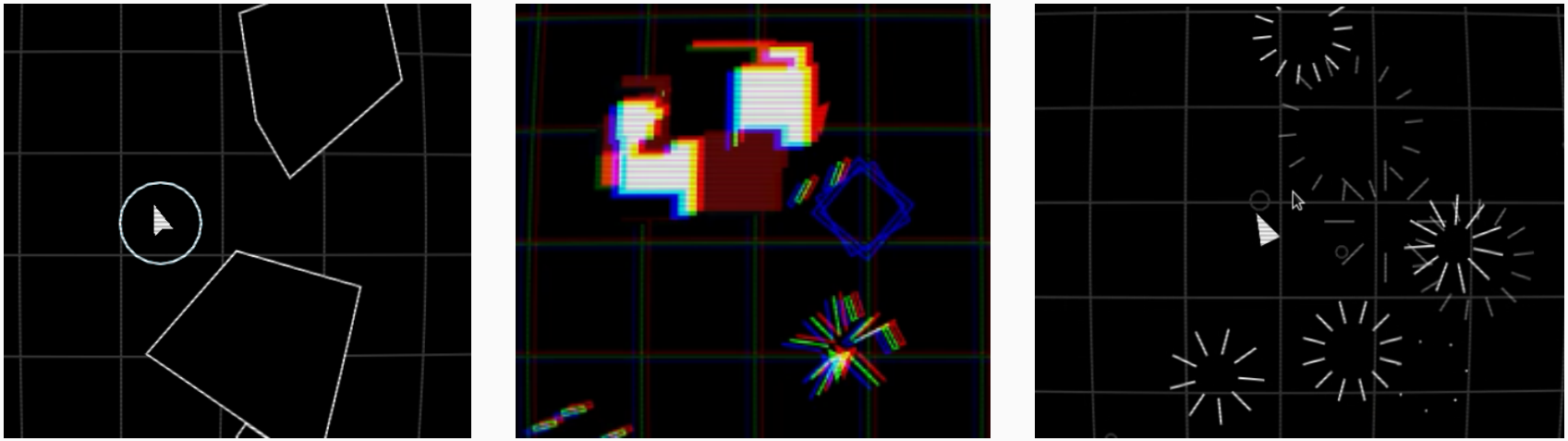
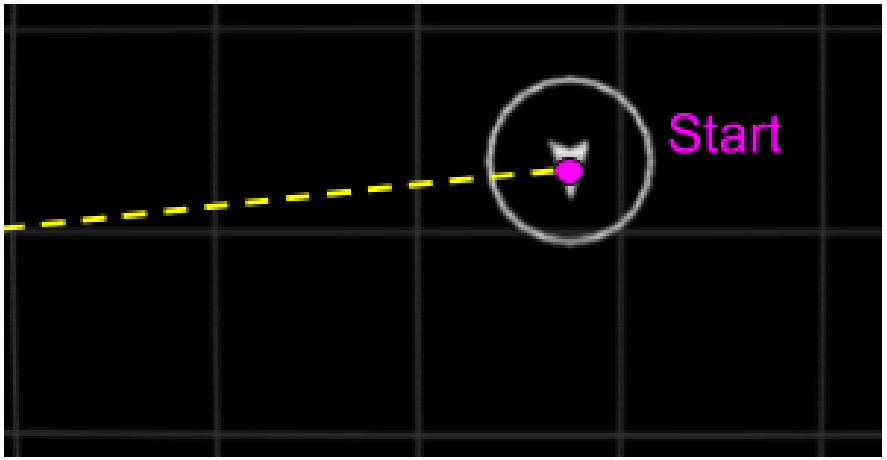


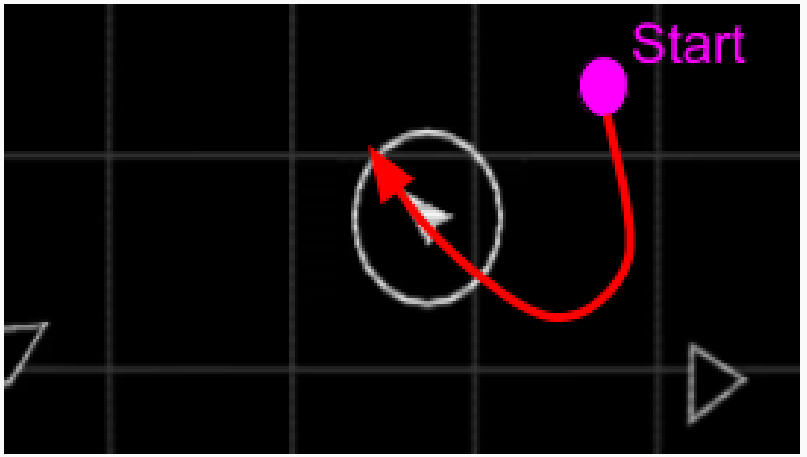
Figure 6: Procedurally generated content

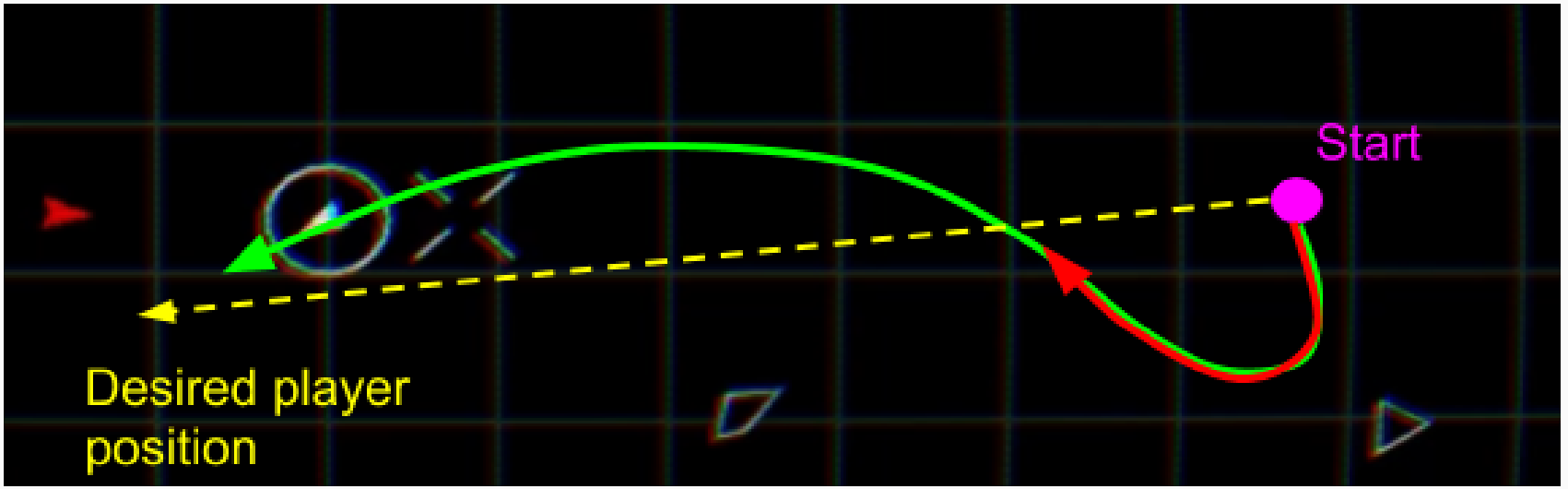
All explosions, asteroids, particle effects and HP/ammo/fuel/oxygen pickups are procedurally generated and placed throughout the world.

Artificial Intelligence

Player ship AI uses an algorithm to follow the players cursor by applying forces to the ship. - Natural movement







AI cont.

Enemy AI works in a similar fashion to the Player's, except it follows the players position.

Enemy AI also uses a **Finite State Machine** to decide on what action it should take from: **Idle**, **Chasing**, **Firing** and **Retreat**.

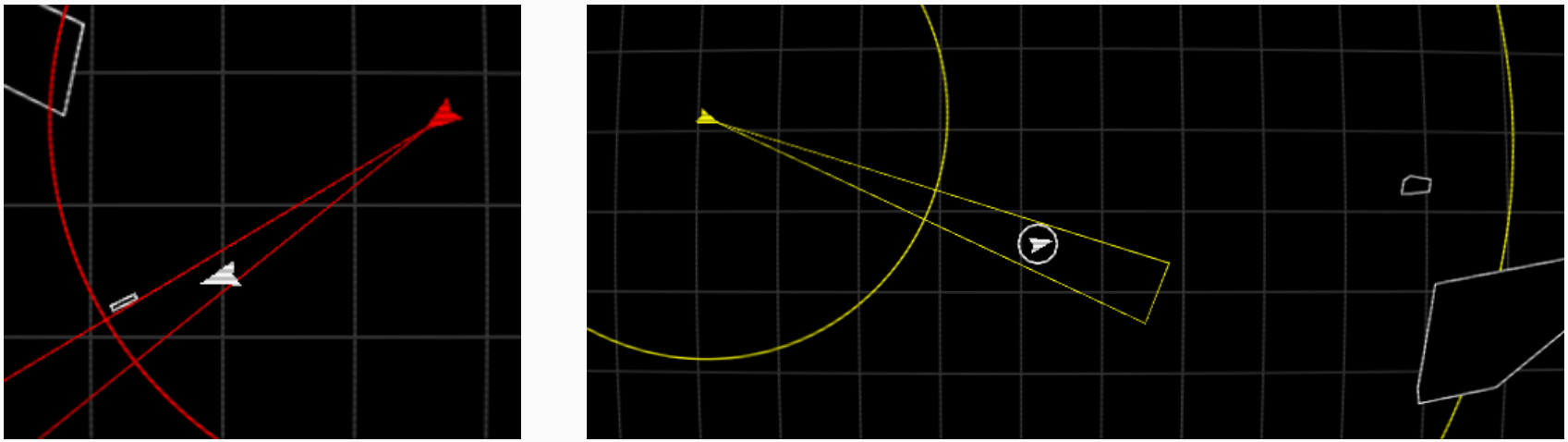


Figure 7: Red: Fire, **Yellow**: Chase

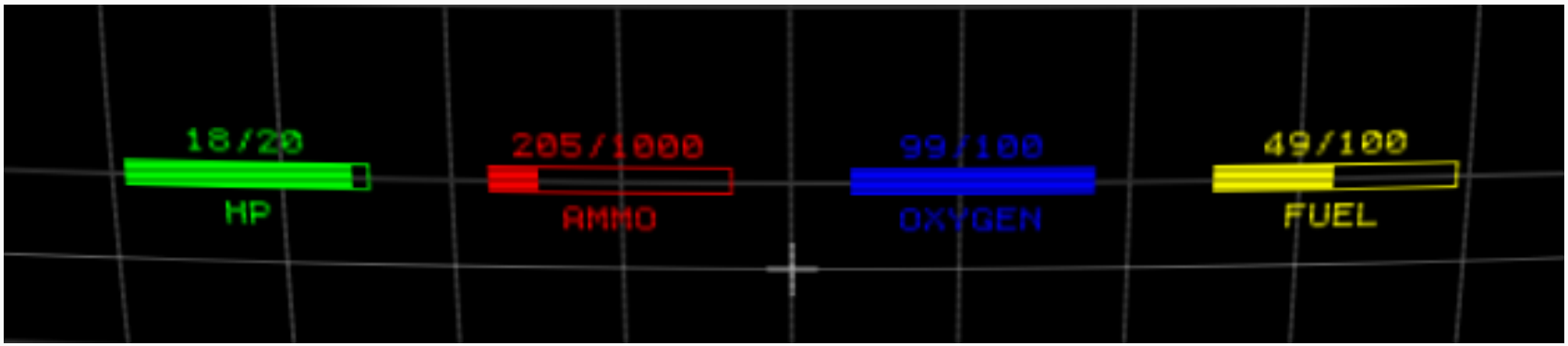
States are decided based on HP, player distance and object count in the world.

Resources, HUD (Heads-Up-Display)

Displays player's Ship details, introduces a layer of difficulty for player to manage.

- **Ammo**, no longer able to shoot
- **Fuel**, speed significantly reduced
- **Oxygen**, lose HP over time
- **HP** (Health Points), when = 0, game over

These can be restored by collecting pickups scattered throughout the world.



Timers and ”tweening”

- **Timers**, allows for events to be repeated / done at specific intervals
- **”Tweening”**, In-be*tweening*, modify a value over time with different easing functions

```
-- Toggle light on and off every second
Timer.every(1, -> lamp\toggleLight())

-- Moves "ball" object to the position 200, 300
 ↪  over 4 seconds
flux.to(ball, 4, { x: 200, y: 300 })
```

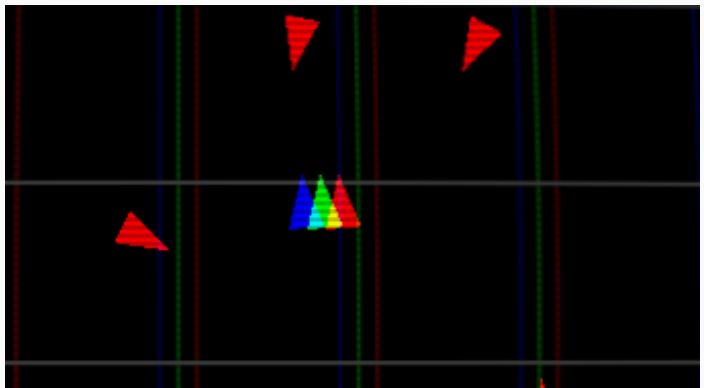
Timers used to spawn new enemies and pickups in the world. Tweening largely used in UI, e.g. particle effects, explosions etc.

Shaders and Cameras

- **Shaders**, post-processing effect that modifies the attributes of pixels, e.g. blurring, shadows and specular highlights.
- **Cameras**, allows for panning, zooming and scaling

e.g. When player hit by a bullet the following occurs,

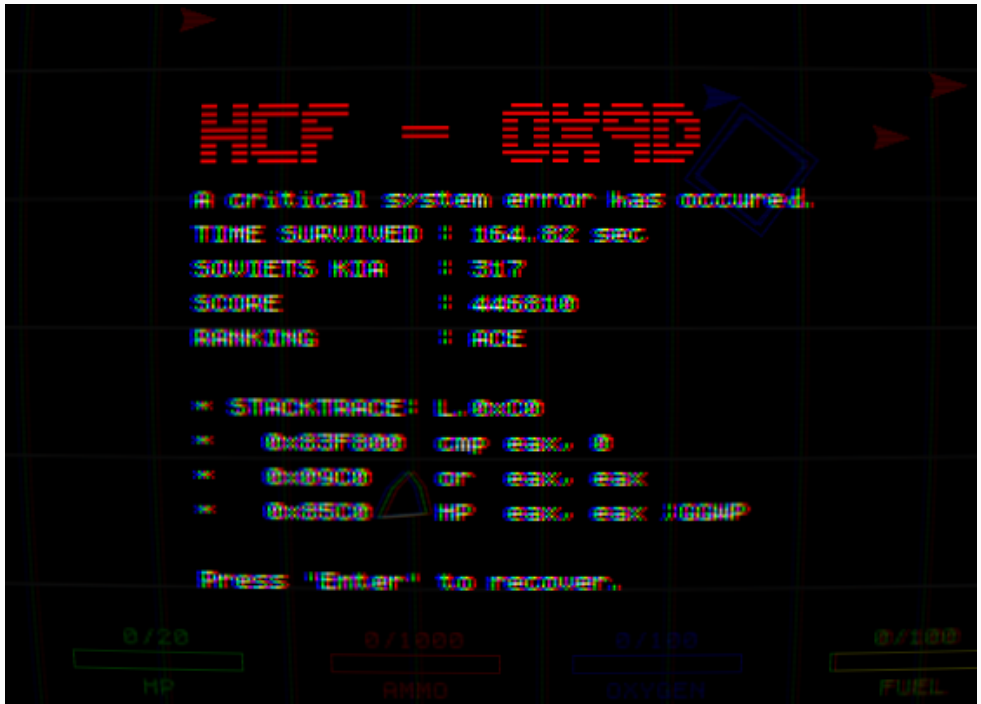
- Instance of class **LineExplosion** at player x/y created
- Tween chromatic aberration strength to random value
- Shake screen by *x* amount for *y* seconds
- Blink screen for $0.1 \pm x$ seconds



Scoring

Higher score is better.
Staying alive for longer grants higher score.
Destroying more enemies/asteroids and collecting pickups increases overall score.

A rank is calculated from overall score:
ACE, SS, S, A, B, C, D, E, F

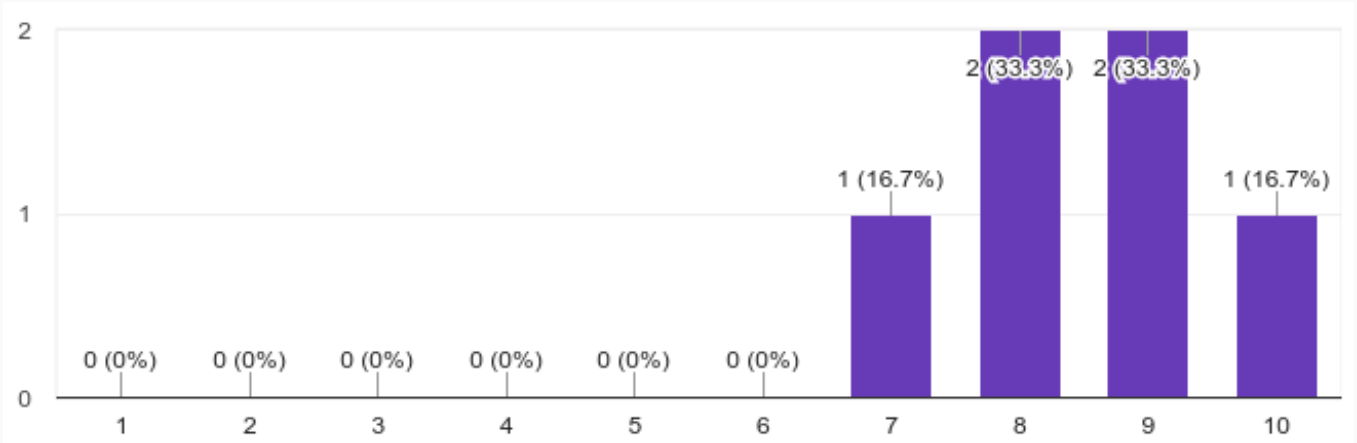


Playtesting and Feedback

Feedback gathered from 8 play-testers on what they liked/disliked and would like to see added/removed. Common positive/negative feedback included:

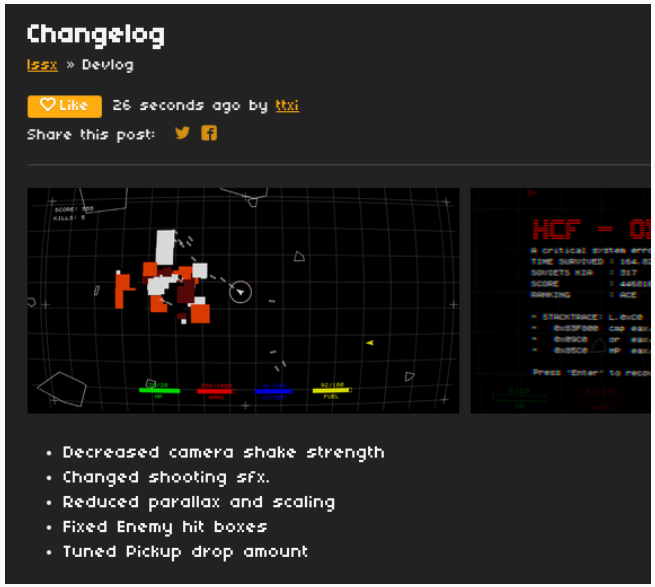
- ”well thought-out aesthetic”
- ”fast, enjoyable pace and high learning curve”
- ”annoying, jarring shooting noise”
- ”too much camera shake”

Players rated game in current state an average of *8.5/10*.



Release

After resolving previous issues with the game, **lssx** was now ready for release.
Released for free with an optional donation feature on **itch.io**, a website for users to host, sell and download indie video games.



Live at : `https://ttxi.itch.io/lssx`

Final thoughts

Overall pleased with outcome, achieved my goal of making a short enjoyable game with a fair amount of re-playability. Gained a strong understanding of *Box2D* and appreciation for simple, elegant solutions to problems.

Things learned:

- Physics is hard
- I don't like gamedev as much as I thought
- Desire to work on a project drops off exponentially
- 90% effort required for the last 10% of work

Areas I'd like to improve:

- Improve *zephyr*
- Add more enemy/weapon types

Special thanks to,

- **SSYGEN**, STALKER-X camera library
- **rxi**, flux.lua tweening library, lovebird browser console
- **vrld**, HUMP utilities, moonshine post-processing library
- **videah**, splash-screen library
- **Taehl**, sound management library
- **bfxr**, used to generate game audio
- **slime et al.**, LÖVE framework
- **leafo**, MoonScript programming language

`https://github.com/twentytwoo/lssx`
`https://github.com/twentytwoo/zephyr`

All software released under MIT license. - LaTeX