lssx, LOVE Space Shooter X

JLC

2018

Project aims

Aims are to create a casual space simulation shoot-em-up with semi-realistic physics and retro graphics, akin games in the 1st/2nd generation of console, circa. 80's

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

Inspiration

- BYTEPATH
- Reassembly
- DATA WING

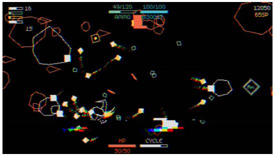


Figure 1: BYTEPATH

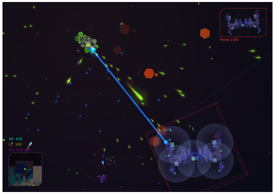


Figure 2: Reassembly

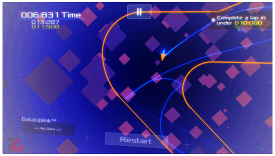


Figure 3: DATA WING

Early computer graphics and design, similar to the Bell Labs / UNIX aesthetic. Cold war paranoia, fantasy systems (Star Wars ICBM defence) - Training simulator for cold-war pilots.

Architecture

LOVE, 2D game development framework, provides interface between code and graphics.

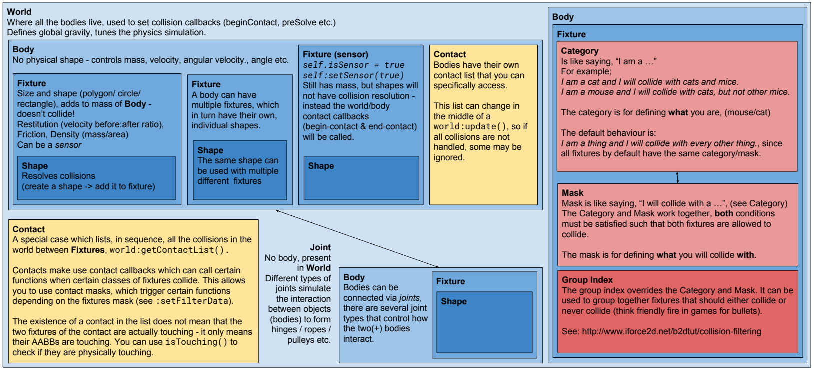
```
function love.draw()
  love.graphics.print("Hello World!", 400, 300)
end
```

Box2D, 2D physics engine for simulating the interaction between rigid bodies, love.physics

```
function someShape:new(x, y, w, h, type, density)
  self.body = love.physics.newBody(world, x, y, type)
  self.shape = love.physics.newRectangleShape(w, h)
  self.fixture = love.physics.newFixture(self.body,
   ↪  self.shape, density)
end
```

Box2D

Box2D is a highly tested, reliable and complete 2D physics engine that powers almost all interactions within the game.



World
Where all the bodies live, used to set collision callbacks (beginContact, preSolve etc.)
Defines global gravity, tunes the physics simulation.

Body
No physical shape - controls mass, velocity, angular velocity., angle etc.

Fixture
Size and shape (polygon/ circle/ rectangle), adds to mass of **Body** - doesn't collide!
Restitution (velocity before:after ratio), Friction, Density (mass/area)
Can be a *sensor*

Shape
Resolves collisions
(create a shape -> add it to fixture)

Fixture
A body can have multiple fixtures, which in turn have their own, individual shapes.

Shape
The same shape can be used with multiple different fixtures

Fixture (sensor)
`self.isSensor = true`
`self:setSensor(true)`
Still has mass, but shapes will not have collision resolution - instead the world/body contact callbacks (begin-contact & end-contact) will be called.

Shape

Contact
Bodies have their own contact list that you can specifically access.

This list can change in the middle of a `world:update()`, so if all collisions are not handled, some may be ignored.

Contact
A special case which lists, in sequence, all the collisions in the world between **Fixtures**, `world:getContactList()`.

Contacts make use contact callbacks which can call certain functions when certain classes of fixtures collide. This allows you to use contact masks, which trigger certain functions depending on the fixtures mask (see `:setFilterData`).

The existence of a contact in the list does not mean that the two fixtures of the contact are actually touching - it only means their AABBs are touching. You can use `isTouching()` to check if they are physically touching.

Joint
No body, present in **World**
Different types of joints simulate the interaction between objects (bodies) to form hinges / ropes / pulleys etc.

Body
Bodies can be connected via *joints*, there are several joint types that control how the two(+) bodies interact.

Fixture

Shape

Body

Fixture

Category
Is like saying, "I am a …"
For example;
I am a cat and I will collide with cats and mice.
I am a mouse and I will collide with cats, but not other mice.

The category is for defining **what** you are, (mouse/cat)

The default behaviour is:
I am a thing and I will collide with every other thing., since all fixtures by default have the same category/mask.

Mask
Mask is like saying, "I will collide with a …", (see Category)
The Category and Mask work together, **both** conditions must be satisfied such that both fixtures are allowed to collide.

The mask is for defining **what** you will collide **with**.

Group Index
The group index overrides the Category and Mask. It can be used to group together fixtures that should either collide or never collide (think friendly fire in games for bullets).

See: http://www.iforce2d.net/b2dtut/collision-filtering

Figure 4: Anatomy of Box2D

LOVE uses an implementation of Box2D called *love.physics* which is essentially the same - but in Lua.

Object Oriented Programming

Object Oriented Programming is a programming paradigm that attempts to deconstruct complex objects into simple components that follow a *parent-child relationship*, for example:

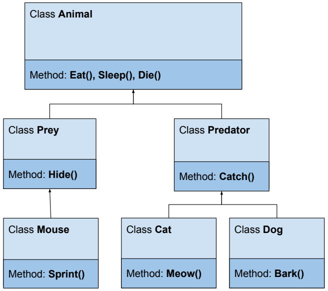


Figure 5: Basic OO structure

Objects have classes, parents, values and methods. This allows an OO system emulate encapsulation, polymorphism and inheritance.

Choosing an OO library

Since *Lua* doesn't natively support OO without the use of *metatables* and *metamethods* (which becomes incredibly verbose quickly) - the *Lua* community has created a series of libraries which simplify the use of OOP.

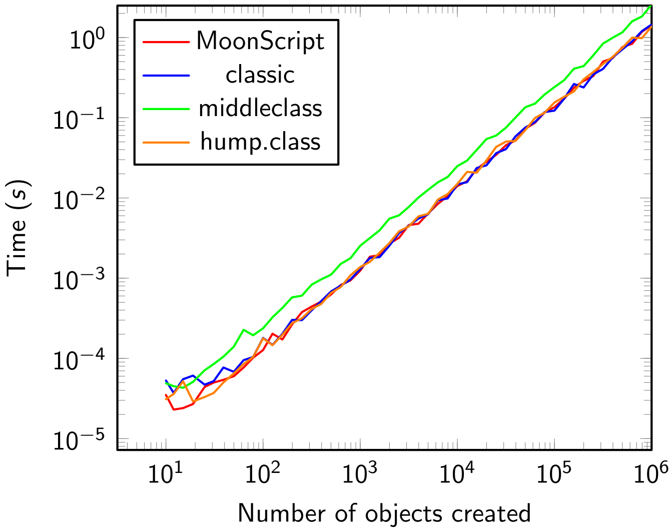
A series of tests was performed to see which library was the most memory efficient and fastest.

classic, *hump.class*, *middleclass* and a language that compiles into *Lua* called *MoonScript* were all tested using a small program which recorded time taken for each library to:

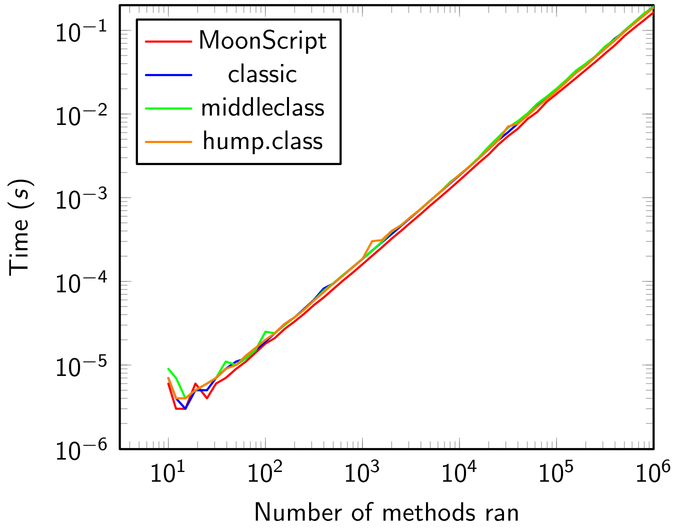
- Create objects
- Perform methods
- Create objects with a parent (test inheritance)

From 10 to 1 million objects.

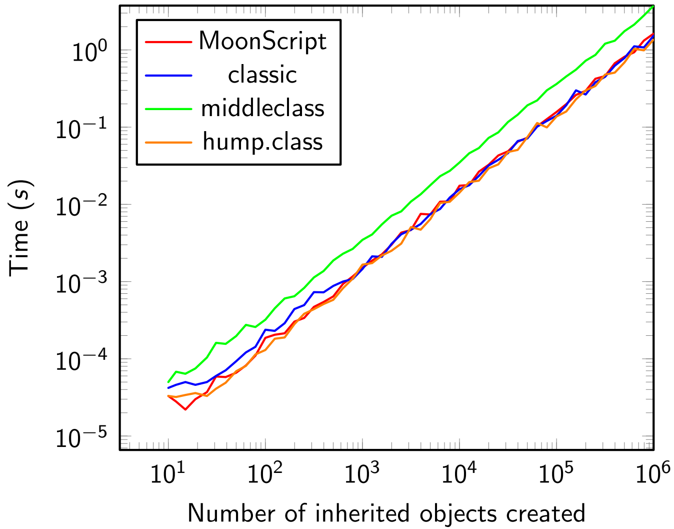
Creating objects



Performing methods



Testing inheritance



Results and conclusion

The results show a close tie with *classic* and *hump.class* - *middleclass* lags behind because of the quantity of features it has.

MoonScript is the leader with a small amount of objects (< 100).

MoonScript was chosen to be the programming language for this project because it applies a level of abstraction above Lua which means less code is required to write the same thing. The compiling process allows for adjustments to be made to the code which allows it to run faster prior to the program running.

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
```

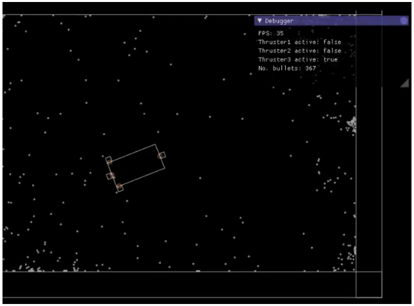
Total lines of code required to do the same thing is drastically reduced, overall code readability increases and the chances of errors arising from syntax is also reduced.

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
- *Speed*, the engine should run quickly
- *Readability*, the code should be easy to read, with most contents' operation being understandable at first-viewing

Rapid prototyping

A small prototype which used a majority of Box2Ds features was created in under a week to test if the project was feasible.



```
FPS: 35
Thruster1 active: false
Thruster2 active: false
Thruster3 active: true
No. bullets: 367
```

Figure 6: Prototype

This prototype was created prior to the entity management system and object orientation, as a result it had to be completely reworked as it was unscalable.

zephyr

Mid-way through the project I realised the current method of detecting and handling collisions involved a lot of repeated code and was generally un-scalable.

zephyr is a Box2D wrapper designed to make the process of creation easier by streamlining collision detection and resolution between Box2D objects.

Collision detection for all objects is now done simply;

```
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 incredibly fast object identification with the use of **Universally Unique IDentifier**'s organised in a hashtable.

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

Objects in the game can be accessed by their **UUID** via,
`lssx.objects[UUID]`

Physics fixtures have their **UUID**'s referenced in the fixture `UserData`, such that when two fixtures collide, their `UserData`'s act as pointers to the object in the global object table, `lssx.objects`

zephyr and Entity Management cont.

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

```
7.365s [collision ] -> Asteroid, k:
 ↪  a49d3d23-42a8-4a55-8f37-f1dbdea8bda4
7.365s [collision ] -> Bullet,   k:
 ↪  a0730861-a8ad-4133-b09b-bacb7150f030
```

This shows a collision between an *Asteroid* and *Bullet*, with their **UUID**'s defined as `k`, each object was found within less than 0.001 second of each other.

Procedural Generation

Creating a large quantity of content on my own would take a significant amount of time and effort. Procedural Generation is a method of computationally generating content.

For example, an *Asteroid* is a polygon shape with a somewhat random convex shape:

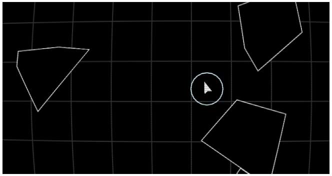


Figure 7: Procedurally generated asteroids

Asteroids also break apart when hit by projectiles, the broken down asteroid shapes are also procedurally generated via algorithms.

AI

AI works in a similar fashion to the Player/'s ship, which follows the mouse position. An algorithm roughly calculates the shortest path from itself to the player, and a

HUD, Heads-Up-Display

Scoring

Gameplay link

https://youtu.be/eq3moJlIBcQ