Physîsîm

Physics Simulator and Interpreted Scripting Language

Brendan Haney

12/14/2024

INTRODUCTION

When designing a programming language, it is essential to consider a specific problem that the language aims to solve. The goal should be to clearly identify the task at hand and design a solution that helps achieve it more efficiently. The first programming languages were created to abstract the process of writing binary code, allowing computer code to become human readable and eliminating the need to manually write machine code. This concept of abstraction can be applied to more than just writing code for computers. For example, when solving physics problems in school, how can you effectively visualize the concepts that are being represented on paper? Similarly, when observing physics simulators (impressive programming feats in their own right), how do you precisely define the scenarios you want them to simulate?

Introducing *Physisim*, a dual-system that combines a graphical physics engine with a custom scripting language. The scripting language (dubbed *Physiscript*) allows users to define interactions within the simulation, while the physics engine handles the underlying calculations. Both the physics engine and the interpreter are complex systems in their own right, each presenting its own set of challenges. Together, they form the foundation of this project.

BACKGROUND

At a high level, *Physisim* can be understood as the union between a graphical physics engine and an interpreter. Both of these major components are the heart and soul of *Physisim*, and each of these two components have a multitude of sub-components. If one wishes to replicate this project, it is crucial that they understands the different sub-components that go into creating a project such as this. A physics engine in of itself is a complicated amalgamation of various sub-components, each of which present their own challenges in creating. On top of this complexity, *Physisim* includes it's own custom interpreter and graphics engine.

The sub-components of each major part of *Physisim* can be visualized as such:

Physiscript Interpreter

- Lexer
- Tokenizer
- Token dictionaries
- Current-operation next-operation (CONO) table
- CONO pairs & operations

Physics Engine

- Engine itself (handles time intervals)
- Display & graphics engine
- Collision handler
- Object & world classes
- Vector math utility functions

MOTIVATION

The motivation behind this project stemmed from a desire to explore the combination of programming languages and real-world simulation. Physics engines are powerful tools for simulating complex systems, but they often require specialized knowledge to effectively interact with. By creating a custom scripting language, the aim was to simplify the process of defining and manipulating physical interactions, making it more accessible to users without a deep technical background to create their own scenarios and watch them play out in real time.

REQUIREMENTS

The *Physisim* project is built upon two main components: the *Physiscript* Interpreter and the Physics Engine. Each of these components has specific sub-components that define their functionality and contribute to the overall operation of the system.

Physiscript **Language**

The scripting language is designed to be written with the same logic that physics problems are written with on paper. The language should be able to translate into english when read out-loud.

Writing Physiscript

Objects and forces are defined simply by calling the definition and then listing it's identifier. The definition "PARTICLE obj1" will do exactly as it sounds: declare a particle called "obj1".

When it comes to parameters, they should be able to be applied how you would describe an object in real life. For example, if you wanted to describe the mass of "obj1", one would say:
"obj1's mass is 14 kg".

Translating this to P*hysiscript*:

```
obj1 MASS 14
```

Any parameter with the tag (vector format) needs both an x and y floating point parameter. These are defined in parenthesis as such: (x, y).

For example:

```
f1 MAG (-2.3, 7.9)
```

This will set a vector magnitude of x = -2.3 N and y = 7.9 N to the force defined as f1.

Definitions

PARTICLE – define a spherical particle vector
FORCE – define a force vector

Parameters

MASS – set object's mass
RADIUS – set how big a spherical object is
DENSITY – sets how dense an object is
ELASTICITY – sets the "bounciness" of an object
POS – set object position (vector format)
MAG – set a force magnitude (vector format)

Events

APPLY – sets a time stamp for defined force to be applied to an object. Use the '@' symbol to define timestamp.

Example event:

```
APPLY f1 obj1 @ 10
```

"apply f1 to obj1 at 10 seconds…"

All numerical values in Physiscript are assumed to have base units. This means that one should be familiar with what unit a parameter is recorded in and make their definitions accordingly. This design choice was made to simplify the language and make it easier to read, as every physics student should be familiar with units.

Physiscript **Interpreter**

Lexer

The lexer is responsible for splitting the input buffer into individual tokens. This is accomplished using "delimiters", or characters that determine when to 'cut' the string. The lexer written for Physisim uses the following delimiter string:

```
static char DELIMITERS[] = " \n\t";
```

Most programming languages provide functionality to split strings given a set of delimiters. Because Physisim is written in C++ (and originally C), it has a custom parser that deals with this. All that is required of the lexer for the purposes of this project is the ability to return an array of strings that have been parsed in accordance with the above delimiters.

Tokenizer & Encoding

The tokenizer processes the token array produced by the lexer, classifying them into recognizable types that can be interpreted by the system. It must accurately identify keywords, operators, variables, and literals, and ensure the integrity of the token stream for further processing. The tokenizer stores each token string in a dictionary, and then adds it's index into an integer array. This process is called encoding. After encoding, this integer array is returned and used for further processing.

Token Dictionaries

For this interpreter implementation, dictionaries are used to store string tokens. This is important for optimizing space (with repeat tokens) and allowing efficient retrieval of information given an index. These dictionaries map each token to its corresponding data structure, allowing the interpreter to efficiently reference and process tokens when evaluating CONO-pairs. This structure must be dynamic and support quick lookups to ensure smooth execution. *Physisim* has a custom dictionary data structure designed for the *Physiscript* language.

CONO Pairs & Operations

CONO pairs define the relationships between operations, while operations themselves represent the executable actions. When two token indexes are inserted into a completed CONO table, an operation is returned. By traversing through the token index array and gathering CONO pairs, it is possible to execute actions in-sequence of one another. This is the core of how the interpreter works.

Current-Operation Next-Operation Table

The CONO table defines how operations are handled in sequence, based on the current operation and the next operation to be executed. It serves as a critical part of the logic that dictates the flow of operations during script execution. The table itself is defined before processing. This is necessary to ensure a consistent application of CONO pairs.

Physiscript CONO Operations

ERR – Fail-safe in case of incorrect indexing.
NOOP – do nothing.
INIT – begin initialization stage of script (unused)
EVENT – begin event stage of script (unused)
APPLY – schedule an event with timestamp
DEFINE – define a new variable given identifier
PARAM – set object parameters given identifier

Putting it all together, we are left with a fully fledged scripting language interpreter. Below is the code implementation of all these components together, as well as the core “execution” function in which you can see how all of the data is handled by the CONO operations.

Interpreter Class & Event Structure

```
typedef struct {
  int fi;
  int oi;
  int time;
  bool done;
} event;

class Interpreter {
  private:
    void execute(int co, int no, int last);

    char *buffer;                       //file buffer
    dictionary_t dict;                  //language dictionary
    char **tokens;                      //token array (strings)
    int *codes;                         //encoded array (integers)
    int **cono;                         //CONO table
```

CONO Operation Execution

```
void Interpreter::execute(int co, int no, int last) {
  int op = cono[codes[co]][codes[no]];
  int index;
  int li;
  int c = 0;
  char *str;
  char **vecstr;
  const char *vecdelims = ",";
  float temp = (float)codes[co+1];
  event ev;
  switch(op) {
    case APPLY:

      ev = {
        table_get_i(vectbl, codes[co+1]),
        table_get_i(objtbl, codes[co+2]),
    std::map<std::string, int> kmap;  //CONO memory during execution
    std::map<std::string, int> smap;  //CONO memory during execution

  public:
    void read_file(char *filepath);    //read file from disk
    void run_code();                   //interpreter main function

    Interpreter();

    int objc;
    int vecc;
    int eventc;
    table_t objtbl;
    table_t vectbl;
    std::vector<Object> objs;
    std::vector<Vec>    vecs;
    std::vector<event>  events;

};
```

Interpreter Main Function

In sum, the interpreter functions as such: read file → split string buffer → tokenize strings → encode tokens → march through token array & insert into CONO table → execute operations.

```
void Interpreter::run_code() {
  if (buffer == NULL)
    return;
  int count = 0;
  char *delims = DELIMITERS;
  char **lines = split(buffer, delims, &count); // split string buffer
  int codec = count;
  codes = encode(lines, &codec, &dict); //encode string buffer

  for (int i = 0; i < codec; i++) {         //march through the cono table
    int co = i;
    int no = march_ops(&dict, codes, codec, i);
    if (no < codec && cono[codes[co]][codes[no]] != ERR &&
cono[codes[co]][codes[no]] != NOOP) {
      execute(co, no, lastop);            //if cono pair, execute.
      lastop = co;
    }
    i = no-1;
  }

  free(codes);
  free_split_string(lines, count);

}
```

```
      (int) dict_get_f(dict, codes[co+4]),
      false
    };
    printf("ind: %d\n", ev.fi);
    events.push_back(ev);
    eventc++;
    break;
  case DEFINE:
    index = codes[co];
    if (index == 3) { //particle
      Object particle({100,100,0}, 15.0, 0.01, 0.1, 1.8, false);
      objs.push_back(particle);

      table_push(&objtbl, &temp);
      objc++;
    } else if (index == 4) { //force
      Vec force = {10, 10, 0};
      vecs.push_back(force);

      table_push(&vectbl, &temp);
      vecc++;
    }

    break;
  case PARAM:
    index = codes[co];
    if (index == kmap[(std::string)KEY_MASS]) {
      li = table_get_i(objtbl, codes[co-1]);
      objs.at(li).setmass( dict_get_f(dict, codes[co+1]) );
      break;
    }
    if (index == kmap[(std::string)KEY_RAD]) {
      li = table_get_i(objtbl, codes[co-1]);
      objs.at(li).setradius( dict_get_f(dict, codes[co+1]));
      break;
    }
    if (index == kmap[(std::string)KEY_DENS]) {
      li = table_get_i(objtbl, codes[co-1]);
      objs.at(li).setdens( dict_get_f(dict, codes[co+1]));
      break;
    }
    if (index == kmap[(std::string)KEY_ELAS]) {
      li = table_get_i(objtbl, codes[co-1]);
      objs.at(li).setelas( dict_get_f(dict, codes[co+1]));
      break;
    }
    if (index == kmap[(std::string)KEY_POS]) {
      li = table_get_i(objtbl, codes[co-1]);
      str = dict_get_s(dict, codes[co+1]);
      str[0] = ' ';
      str[strlen(str)-1] = ' ';
      vecstr = split(str, vecdelims, &c);
      objs.at(li).setpos({(float)atof(vecstr[0]), (float)atof(vecstr[2]), 0});
      break;
    }
    if (index == kmap[(std::string)KEY_MAG]) {
      li = table_get_i(vectbl, codes[co-1]);
      str = dict_get_s(dict, codes[co+1]);
      str[0] = ' ';
      str[strlen(str)-1] = ' ';
      vecstr = split(str, vecdelims, &c);
      vecs.at(li) = {(float)atof(vecstr[0]), (float)atof(vecstr[2]), 0};
      break;
    }
    break;
  }
}
```

Physics Engine

Engine

The physics engine is the core component of the simulation, responsible for managing time intervals and advancing the state of the simulation. It must support continuous time steps and be capable of simulating real-time interactions with high accuracy. The engine must also provide functionality for managing time-based updates for all objects in the system.

```
class Engine {
  private:
    uint64_t ticks;
    int tps;        //ticks per second
    double interval;
    double dtime;   //milliseconds
    double sdelta;  //seconds
  public:
    void init(char *title, int w, int h, char *fp);
    void update_time();
    void tick();
    void update();
    void check_events();
    uint8_t is_running;
};
```

Display & Graphics Engine

The display engine is tasked with rendering the simulation visually. It must be able to display objects, their interactions, and the environment in real-time, providing clear visual feedback to the user. This engine utilizes OpenGL to render objects. Of the public functions, clean(), draw(), and update() are used in order during the main program loop. The clean() function clears the screen, draw() draws the entire scene to the frame, and update() renders the frame so it can be seen by the viewer.

```
class Window {
  private:
    void drawpixel(float x,float y, int r, int g, int b);
    void drawcircle(float x, float y, float radius, int r, int g, int b);
    void drawline (int x, int y1, int y2, int r, int g, int b);

    int width, height;
    SDL_Window *win;
    SDL_Renderer *renderer;
    SDL_Event event;
    Camera cam;

  public:
    void clear();
    void draw(World w);
    void update();
    void check_updates();
    void kill();

    Window(char *title, int w, int h);
};
```

Collision Handler

The collision handler is responsible for detecting and resolving collisions between objects in the simulation. It must accurately detect when objects overlap or collide and apply appropriate physical responses, which in this case is applying an impulse (reaction force) to each colliding object.

```
bool Collision::circle_int(Object a, Object b) {
  Vec ca = a.getpos(), cb = b.getpos();
  float ra = a.getradius(), rb = b.getradius();
  normal = ZERO;
  depth = 0.0f;

  float dist = distance(ca, cb);
  float radii = ra + rb;

  if (dist >= radii) return false;

  normal = normalize( sub(cb, ca) );
  depth = radii - dist;

  return true;
}

void Collision::resolve(Object *a, Object *b) {
  Vec vrelative = sub(b->velocity, a->velocity);

  if (dot(vrelative, normal) > 0.0f) return;

  float e = min(a->elasticity, b->elasticity);

  float j = -(1.0f + e) * dot(vrelative, normal);
  j /= 1/a->mass + 1/b->mass;

  Vec impulse = scale(normal, j);
  a->velocity = sub(a->velocity, scale(impulse, 1/a->mass));
  b->velocity = sum(b->velocity, scale(impulse, 1/b->mass));
}
```

```
void Collision::handle(Object *a, Object *b) {
  if (a->type == CIRCLE && b->type == CIRCLE && circle_int(*a, *b)) {
    a->move( scale(scale(scale(normal, -1.0f), depth), 0.5f) );
    b->move( scale(scale(normal, depth), 0.5f) );
    resolve(a, b);
  } else return;
}
```

Object & World Classes

The object and world classes define the physical entities in the simulation. The object class must define properties such as position, velocity, mass, and other physical attributes. The world class serves as the container for all objects, managing their relationships and interactions within the environment. Both classes must support efficient updates and retrievals of physical properties.

```
class Object {
  private:
    float angle;
    float vrot;

  public:
    Vec fnet;
    Vec pos;
    Vec velocity;
    float mass;
    float density;
    float elasticity;
    bool equalibrium;
    bool is_static;
    float width;
    float height;
    float area;
    float radius;

    int type;

    void applyforce(Vec force);
    void step(float dtime);
    void move(Vec v);
    void moveto(Vec p);
};

class World {
  private:
    float gravity;
    bool drag;
    std::vector<Object> objects;
  public:
    int type;
    int objc;

    void addobj(Object obj);
    Object *getobj(int id);
    Object copyobj(int id);
    void step(float dtime);
    void handle_collisions();
};
```

Vector Math Utility Functions

The vector math functions provide the mathematical operations needed to simulate motion and force interactions in the physics engine. These functions are crucial when doing physics calculations. They must be able to perform vector mathematics given a vector object “Vec”.

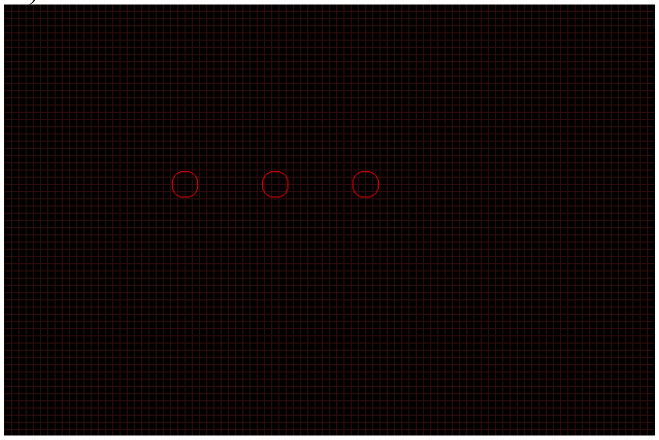
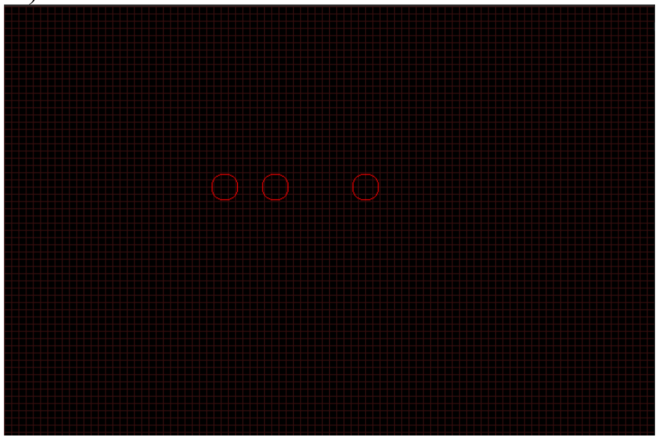
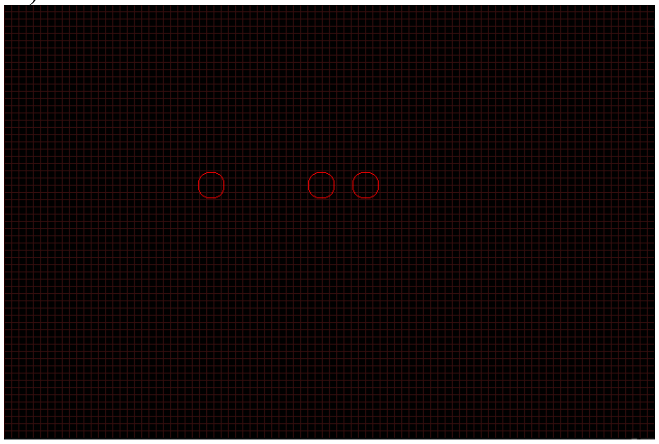
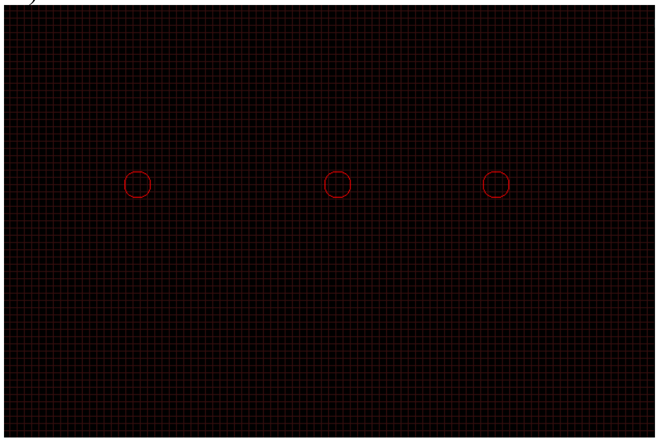
```
typedef struct {
        float x;
        float y;
        float z;
} Vec;

float magnitude(Vec a);
float distance(Vec a, Vec b);
float angle(Vec a, Vec b);
float dot(Vec a, Vec b);
float min(float a, float b);
int fcomp(float a, float b);
int vcomp(Vec a, Vec b);
Vec sum(Vec a, Vec b);
Vec sub(Vec a, Vec b);
Vec scale(Vec a, float s);
Vec cross(Vec a, Vec b);
Vec normalize(Vec a);
```

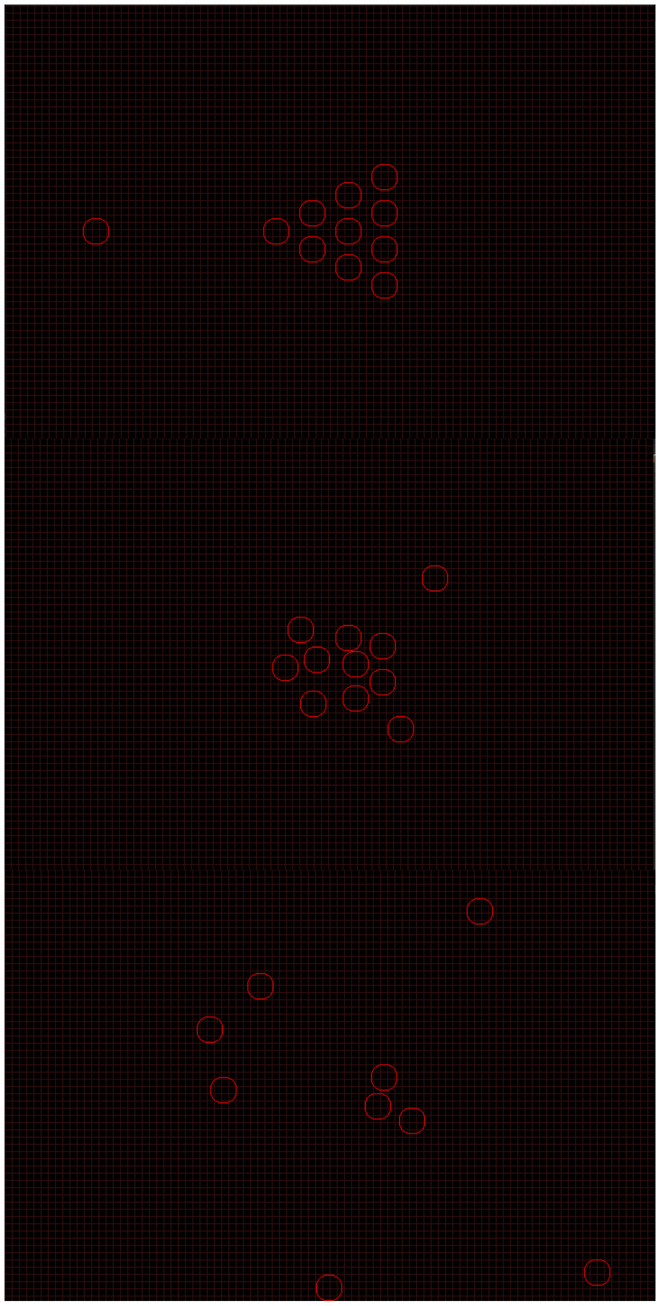
RESULTS
Given the following script:

example.psim

```
PARTICLE a
a MASS 0.3
a POS (200, 200)

PARTICLE b
b MASS 0.3
b POS (400, 200)

PARTICLE c
c MASS 0.54
c POS (300, 200)

FORCE f
f MAG (20, 0)

APPLY f a @ 2
```

We would expect to see (on an X-Y plane) three circles. After exactly two seconds of watching stationary circles, we would expect the leftmost object to begin moving rightward, and collide with the middle object. After this collision, the leftmost object should change it's trajectory, and the middle object should begin moving. Finally, The middle object should collide with the rightmost object, changing the middle trajectory and moving the rightmost. Plugging this script into *Physisim*, we get the following result:

1.) Time < 2 Seconds  Objects in equilibrium.	2.) Time > 2 Seconds  Force applied to leftmost, begins moving.
3.)  Leftmost and middle collide, leftmost moves left and middle moves right.	4.)  Middle collides with rightmost, and the simulation continues.

By writing a more sophisticated script:	We get a more interesting result:
pool.psim PARTICLE white white MASS 0.1 white RADIUS 15 white ELASTICITY 0.4 white POS (100, 250) PARTICLE one one POS (300, 250) PARTICLE two two POS (340, 270) PARTICLE three three POS (340, 230) PARTICLE four four POS (380, 290) PARTICLE five five POS (380, 250) PARTICLE six six POS (380, 210) PARTICLE seven seven POS (420, 270) PARTICLE eight eight POS (420, 230) PARTICLE nine nine POS (420, 190) PARTICLE ten ten POS (420, 310) FORCE f f MAG (20, 0.3) APPLY f white @ 2	

CONCLUSIONS

In conclusion, *Physisim* demonstrates the full potential of designing a custom programming language. By combining a programming language with a graphical physics engine, we simplify and enhance the process of simulating physics scenarios. With *Physisim's* scripting language, *Physiscript*, users can define and manipulate physical systems in a way that is intuitive and closely aligned with real-world physics practice problems. The integration of an interpreter and a physics engine allows for a streamlined process for users with limited technical backgrounds to explore more complex simulations and scenarios, contributing to both educational and professional applications in physics and engineering.

REFERENCES

All project source code can be found at:
https://github.com/pointerisnull/physisim

Physics Engine circle collision math obtained from:
Hecker, Chris. "Physics, Part 3: Collision Response." *Game Developer*, March 1997, pp. 14.
https://www.chrishecker.com/images/e/e7/Gdmphys3.pdf
**Note: nothing was explicitly copied, this source was used for it's mathematical formulas on page 14*