Tidal Tails

April 25, 2017

Abstract

1 Introduction

The aim of this project is to simulate/observe the creation of tidal tails. Tidal tails are created when two galaxies pass one another and interact gravitationally. A tidal tail is an elongated stream of stars that extends outwards from a galaxy. To do this an N-body simulation of massive particles was created using C++, and visualised using SDL2/OpenGL. To observe the formation of tidal tails a galaxy with a fixed central mass was created, test particles were then set in motion in a uniform distribution around this central mass such that the orbits were circular. A perturbing galaxy was then introduced on several different orbits (conic sections) with various input parameters. Tidal tail formation was observed and the results were screenshotted at various times after introduction for each of these orbits.

The computational techniques used in this program will be analysed in section 2, following this the specific implementation and how the simulation performed will be discussed in section 3. The screenshots of the Tidal Tails and discussions of them will be contained in section 4. And finally any concluding remarks will be in section 5. An appendix containing instructions and full code listings is included at the end of this document.

The program created is interactive and the camera can move around the scene using the WASD keys and zoom out/in using the keys Q and E respectively. Screenshots can be taken using the P key (creates a tga image file, can convert to png using the convert_png.sh script provided) and the simulation can be paused/ unpaused using the SPACEBAR (Note pausing will print out the dimensions of the visualisation in the console). Logging of the particle's positions to a text file can be toggled on/ off using the L key. And if the correct flag is activated (INTERACTIVE) masses can be created by clicking (left), dragging and then releasing inside the window (the velocity of the created mass is proportional to the length of the drag). The input parameters for the perturbing galaxy can be set using command line arguments when starting the program (eccentricity, θ_0, Closest Approach, Rotation direction of central galaxy ($-1 = \circlearrowright, 1 = \circlearrowleft$), Perturbation orbit direction). In the simulation the central mass and perturbing galaxy are green in colour, the test particles are red and the trail of the perturbing galaxy is blue.

The aim of my program was to find out how the direction of rotation of the test particle's initial orbit affect the formation of tidal tails for various perturbing galaxy orbits.

2 Analysis of Methods

First of all the problem required scaling, in typical units (SI) the gravitational constant $G = 6.67 \times 10^{-11} m^3 kg^{-1} s$ and a typical central mass might be many orders of magnitude larger than the mass of the sun ($M_\odot = 2.0 \times 10^{30} kg$). So units where G = 1 and the central mass M = 1 were selected (the perturbing galaxy also defaults to having a mass of 1). This scaling is required because otherwise we would have time scales of orbits which are far too long to simulate, and smaller values are in general easier to deal with. Another issue is the potential for infinities to arise in simulations. These infinities arise due to the singular nature of the gravitational force at small distances. To compensate for this the force on a test particle after entering the surface of a central mass was altered to a repulsive radial force ($\frac{GM}{r^2}\underline{\hat{r}}$). In this way a particle can't penetrate far enough in to a particle for infinities to arise. This method provides a crude simulation of collisions between test particles and large masses.

A Verlet integration method was selected for this simulation in part because it is symplectic (unlike RK4) which means it will conserve energy well, this is very important in orbital dynamics to prevent energy drift and to maintain stable orbits. Another reason it was chosen is due to it being only slightly more computationally expensive than other lower order methods (euler) while having errors of order Δt^4 rather than Δt^2. Verlet is also less computationally expensive than RK4. Finally implementing an integrator using Verlet rather than RK4 meant that the acceleration only needed to be evaluated at the particle's current position. The Verlet algorithm requires current and past information only and doesn't depend on predicting future values like RK4. This means Verlet lends itself well to the "real-time" visualisation that this program intends to achieve.

OpenGL/SDL2 were selected to visualise the formation of tidal tails over traditional graphing solutions. This decision was made so that the simulation could be interactive (move around zoom in/out) and also so that tidal tails could be observed continuously and screenshotted during formation in "real time".

The suggested dark matter halo form of the perturbing galaxy (from Project Manual) can easily be implemented by setting the radius of the perturbing galaxy to a size comparable to the central mass plus its test particles and modifying the force function for test particles inside the perturbing galaxy's radius to be the same as for the inside of a uniform spherical mass. $\underline{F} \propto -|\underline{r}_{galaxy} - \underline{r}_{test}|$.

3 Implementation and Performance

The first implementation issue was to decide the integrator to use, Verlet was selected. The Verlet algorithm is as follows

$$\underline{x}_{n+1} = 2\underline{x}_n - \underline{x}_{n-1} + \frac{1}{2}\underline{a}(\underline{x}_n)\Delta t^2. \tag{1}$$

This obviously requires the particle's previous and current position to calculate the particle's next position. At t = 0 we only know the particle's current position so the first time step must be carried out using a different algorithm. A simple 2nd order method was selected to carry out the first time step, $\underline{x}_1 = \underline{x}_0 + \underline{v}_0\Delta t + \frac{1}{2}\underline{a}(\underline{x}_0)\Delta t^2$. All future steps were then carried out using the Verlet algorithm.

The algorithm was tested using circular orbits to determine that it was functioning correctly. The radius of the orbit was plotted as a function of time for several orbits to determine the degree of variation. One of these plots is shown below.

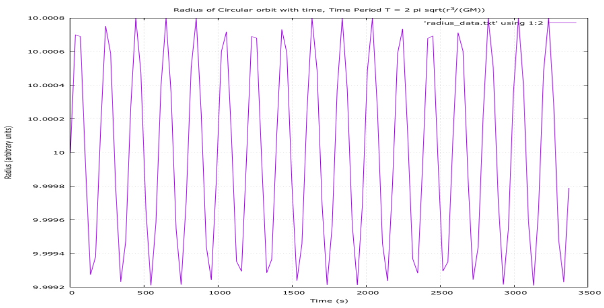


Figure 1: The variation in radius for a test mass in a circular orbit at a radius of 10 (arbitrary units), shows small sinusoidal variations around the initial radius. The graph shows roughly 18 orbits.

The radius shows small order variations, indicating the algorithm was functioning as intended. Testing can be turned on by setting the TESTING flag to true. Orbits for many test particles were observed over many cycles to make sure they were following the expected trajectories (mainly conic sections). These were tested by modifying the galaxy creation step in the orbit_test function in the main source code file.

The loops that determine the accelerations of each of the particles only loop over the particles that are massive, this helps to speed up the computation by removing unnecessary loops. The test particles can be given mass by setting a flag but this results in the breakdown of the galaxies structure over time and also increases the computation time. These effects are undesirable so the test particles were assigned 0 mass. Another optimisation that was used was to store any vectorial quantities (position, velocity, acceleration) in fixed sized arrays of length 3 which are often more efficient to use than dynamic ones. These arrays are 3D so that the program is most general, but they can easily be reduced to 2D by changing the typedef statement of vec3 in the utilities.cpp source file. The results would not change as the visualisation and initial conditions are all in the x-y plane only. This reduction in dimension would reduce the memory footprint of each particle object. But the performance is satisfactory in the general 3D case, so 3D arrays were used.

Performance was monitored to determine any possible causes of slow down. By turning off the rendering portion of the program and logging to data files instead it was seen that most of the computation time was in fact being used to render the particles to the screen after each time step. To reduce the time taken rendering an adaptive frame limiting method was employed. Now particle motion was only rendered after a certain amount of CPU time had passed and not after every time step. Doing this resulted in a much smoother visualisation and allowed the simulation to run much closer to real time. The adaptive portion of this method made it so the number of frames rendered each second scales with the number of particles in the system, meaning larger systems are rendered less often. This helps to prevent the visualisation from slowing to a crawl.

Finally tests were carried out to determine the maximum amount of test particles that could be created without slowing the visualisation too much. With around 8000 test particles the simulations can be completely visualised within a few of minutes (2-5). This seems like a sensible amount of test particles to include. The distributions of these particles were set according to the project manuals guidelines but with many more particles per ring.

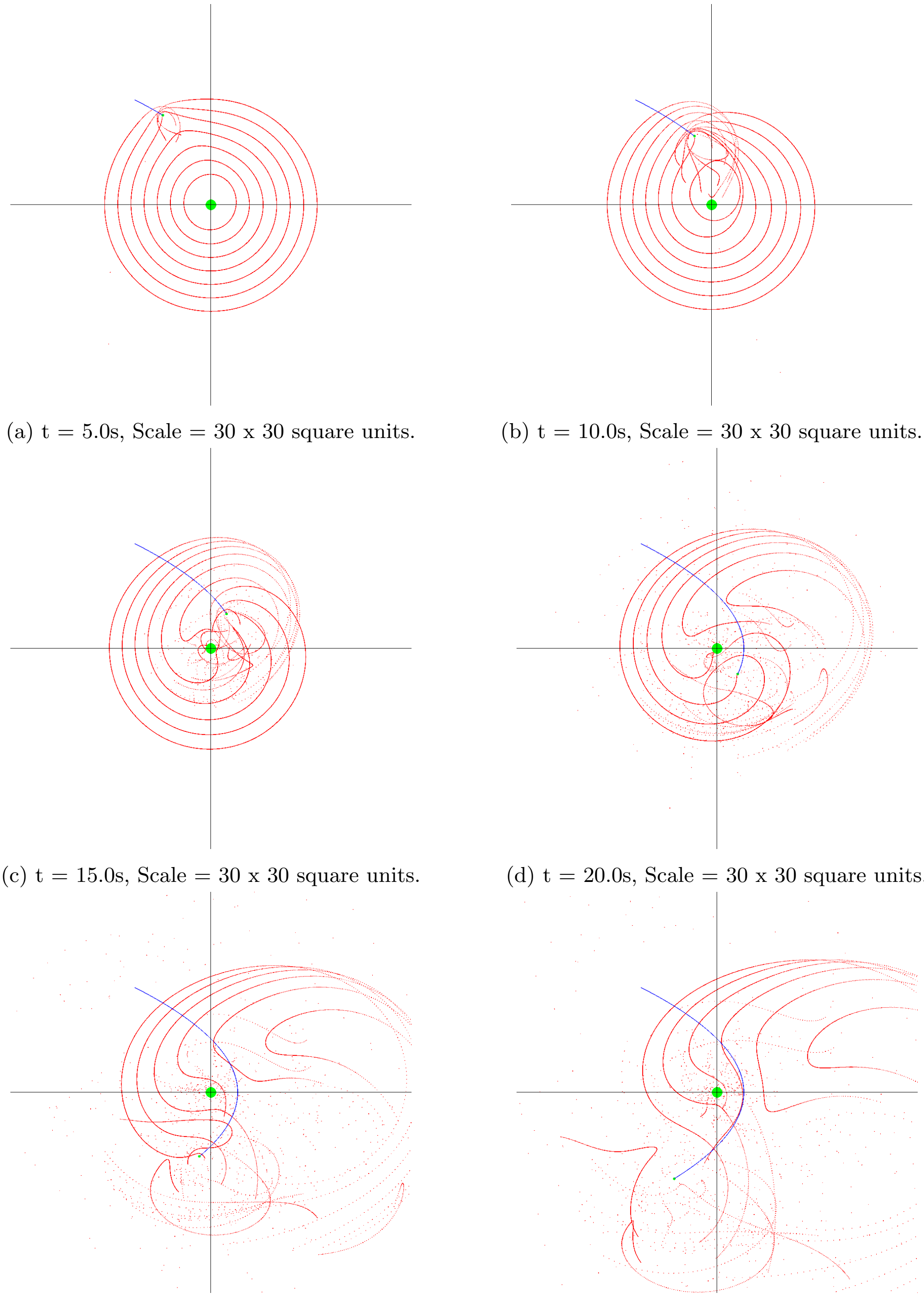
The program was created with object orientation in mind. Four classes were created. A particle class which housed all information about each individual particle and provided a function to render the particle. A universe class which housed pointers to all of the particles in a system, contained functions to generate galaxies, and functions that handled updating particle locations and logging of particle data. A logger class which essentially acted as a data logging device and stored particle positions in a data file when requested. And finally a camera class which stored the camera's location and calculated the zoom factor of the view, these quantities are then used during rendering to project the correct image of the scene.

To simplify the determination of the initial conditions of the system (for a conic orbit) the central galaxy's central mass was fixed at the origin (0,0,0). Choosing a system like this makes it very easy to introduce a perturbing galaxy that orbits in a conic section. The initial conditions for these orbits are quite straightforward to determine using the standard results for conic sections.

1. $E_{total} = -\frac{GM_1 M_2}{2a} = \frac{1}{2}M_1 \underline{v}_1^2 + \frac{1}{2}M_2 \underline{v}_2^2 - \frac{GM_1 M_2}{|\underline{r}_1 - \underline{r}_2|}$
2. Semi major axis $a = \frac{r_{min}}{1-e}$
3. Polar equation of conic section $r = \frac{(1+e)r_{min}}{1+e\cos\theta}$
4. Cartesian coordinates $x = r\cos\theta, y = r\sin\theta$

Combining equation 1 and 2 and using the fact that the central mass (subscript 1) is fixed with 0 velocity at the origin it is possible to find the magnitude of the initial velocity of the perturbing galaxy. To find the direction of the velocity you can use the fact that the velocity vector must be a tangent to the galaxy's orbit. Rewriting equation 3 in Cartesian form and finding $\frac{dy}{dx}$ it is possible to determine this tangent vector. To find the perturbing galaxy's initial position you just need to convert the polar form of the orbit (3) into Cartesian coordinates using (4). To specify these initial coordinates and velocity we need 3 independent parameters the eccentricity e, the starting angle θ_0, and the distance of closest approach r_{min}. These 3 quantities are taken as input parameters in the program and are used to determine the initial conditions for the perturbing galaxy.

4 Results and Discussion



(a) t = 5.0s, Scale = 30 x 30 square units.

(b) t = 10.0s, Scale = 30 x 30 square units.

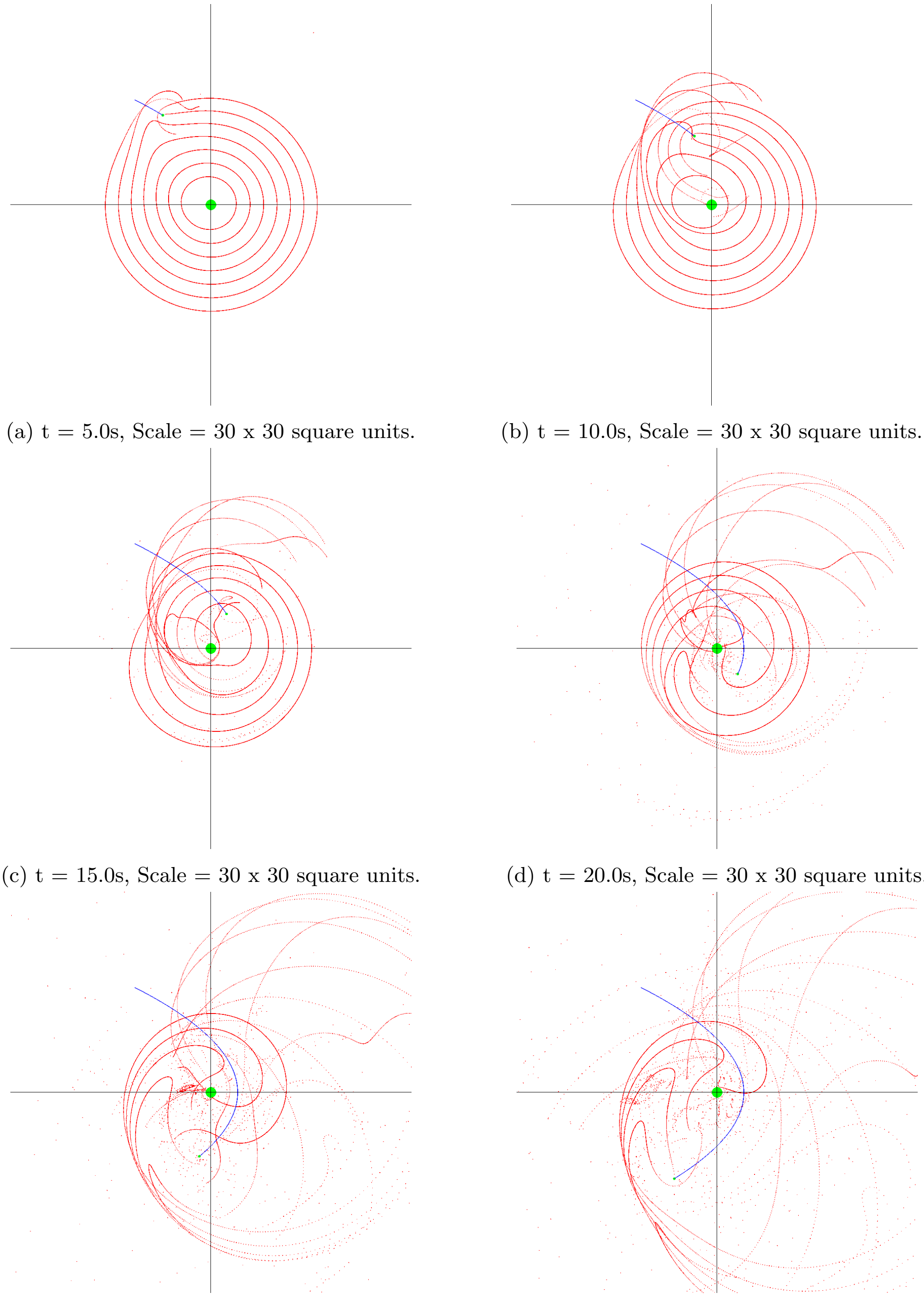
(c) t = 15.0s, Scale = 30 x 30 square units.

(d) t = 20.0s, Scale = 30 x 30 square units.

(e) t = 25.0s, Scale = 30 x 30 square units.

(f) t = 30.0s, Scale = 30 x 30 square units.

Figure 2: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 2 units and starts at an angle of $0.35 \times 2\pi$. Test particles are initially in a clockwise circular orbit around the central mass. (Perturbation direction set to 1)



(a) t = 5.0s, Scale = 30 x 30 square units.

(b) t = 10.0s, Scale = 30 x 30 square units.

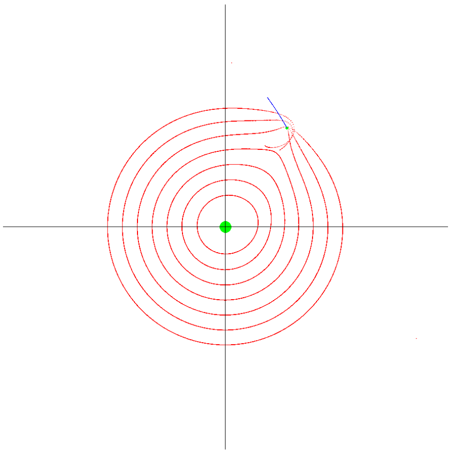
(c) t = 15.0s, Scale = 30 x 30 square units.

(d) t = 20.0s, Scale = 30 x 30 square units.

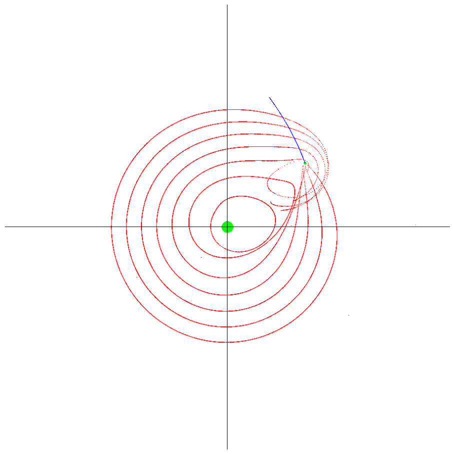
(e) t = 25.0s, Scale = 30 x 30 square units.

(f) t = 30.0s, Scale = 30 x 30 square units.

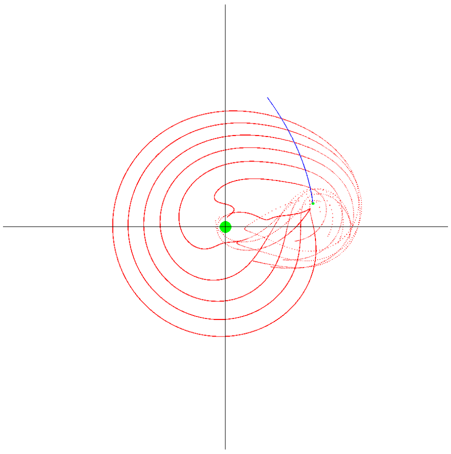
Figure 3: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 2 units and starts at an angle of $0.35 \times 2\pi$. Test particles are initially in an anticlockwise circular orbit around the central mass. (Perturbation direction set to 1)



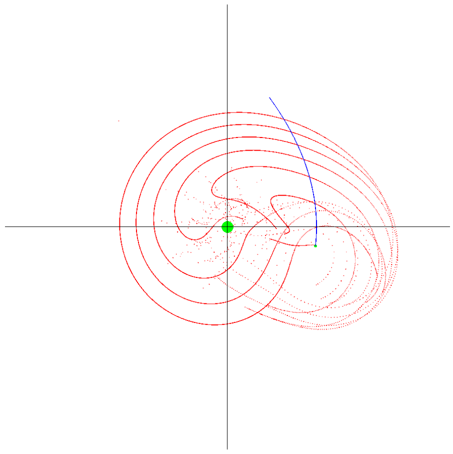
(a) t = 5.0s, Scale = 30 x 30 square units.



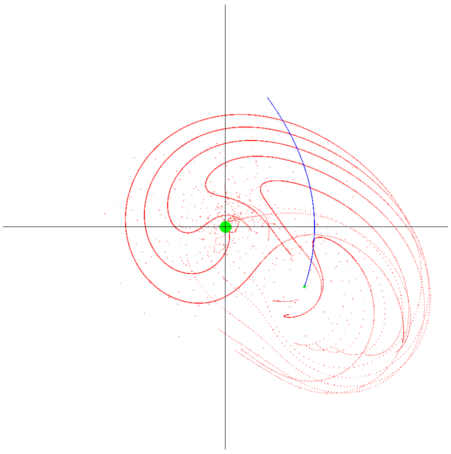
(b) t = 10.0s, Scale = 30 x 30 square units.



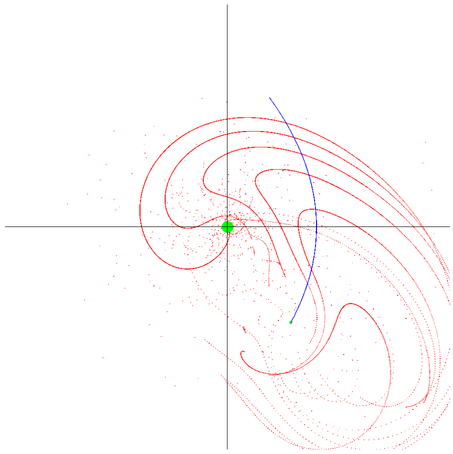
(c) t = 15.0s, Scale = 30 x 30 square units.



(d) t = 20.0s, Scale = 30 x 30 square units.

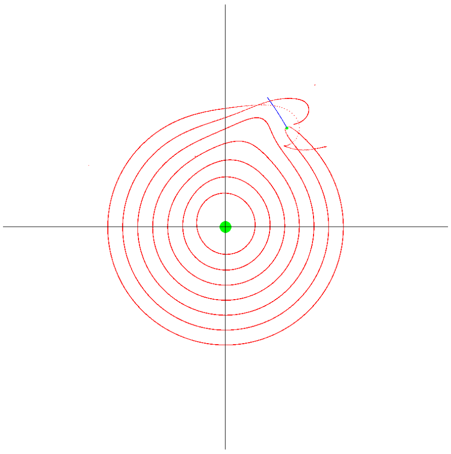


(e) t = 25.0s, Scale = 30 x 30 square units.

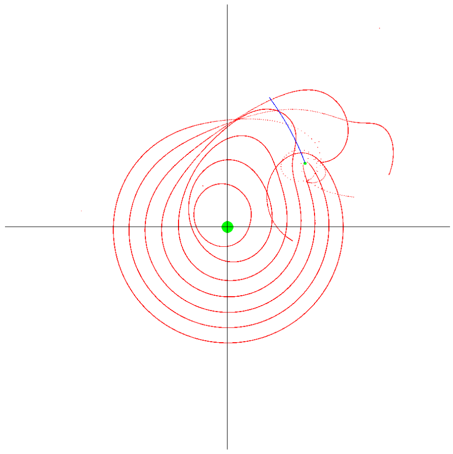


(f) t = 30.0s, Scale = 30 x 30 square units.

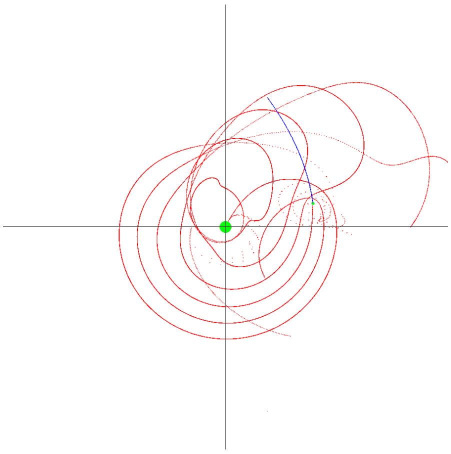
Figure 4: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 6 units and starts at an angle of $0.2 \times 2\pi$. Test particles are initially in a clockwise circular orbit around the central mass. (Perturbation direction set to 1)



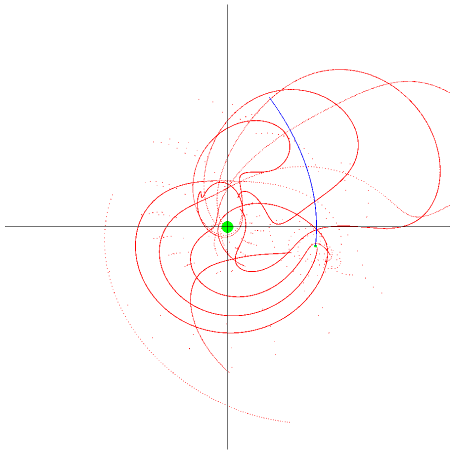
(a) t = 5.0s, Scale = 30 x 30 square units.



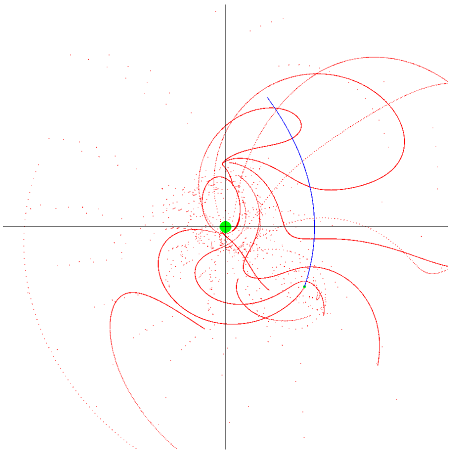
(b) t = 10.0s, Scale = 30 x 30 square units.



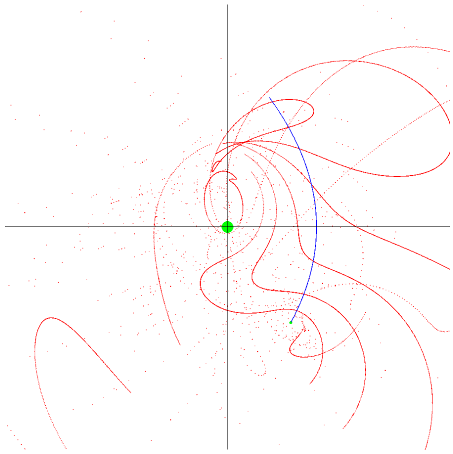
(c) t = 15.0s, Scale = 30 x 30 square units.



(d) t = 20.0s, Scale = 30 x 30 square units.

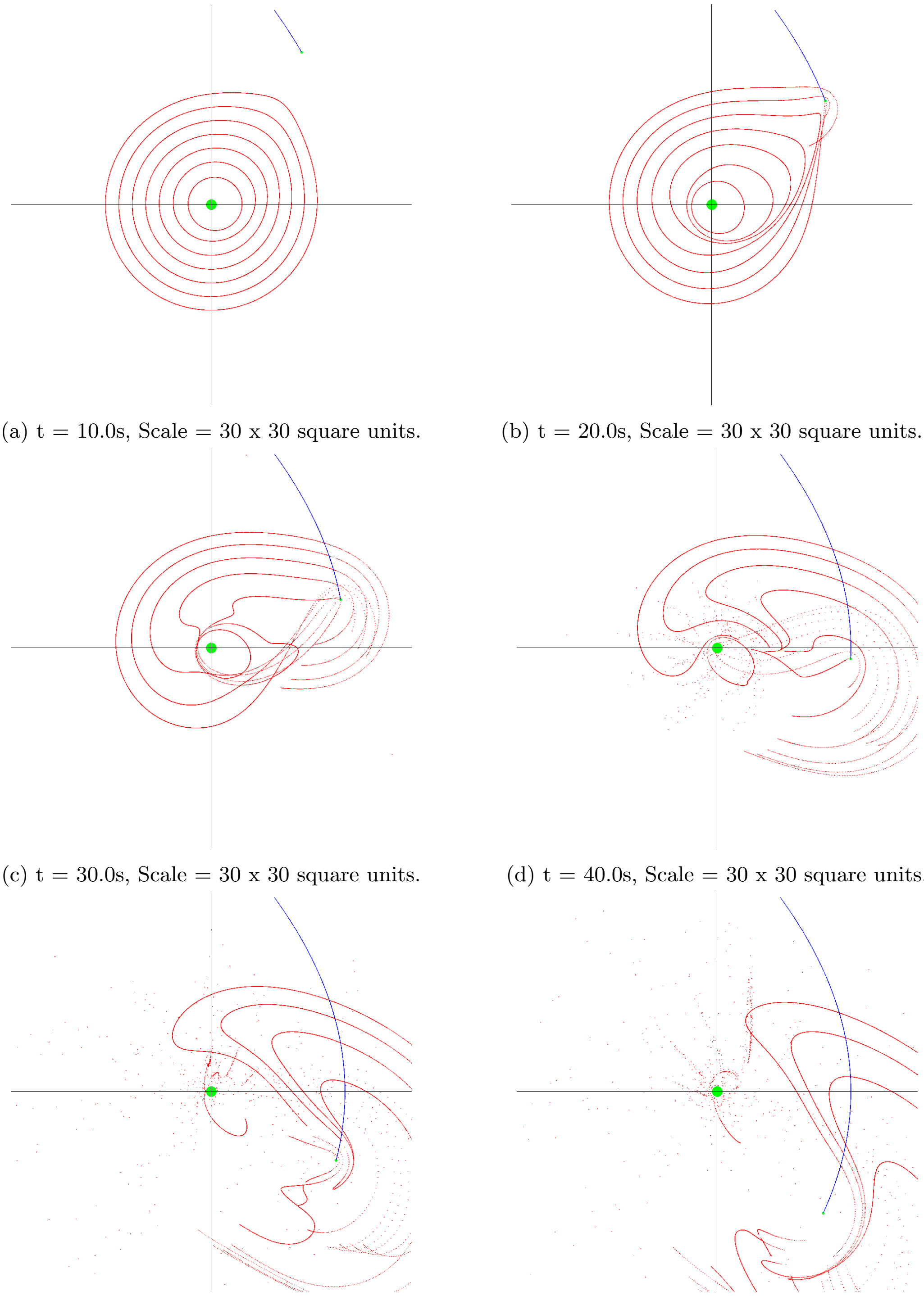


(e) t = 25.0s, Scale = 30 x 30 square units.



(f) t = 30.0s, Scale = 30 x 30 square units.

Figure 5: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 6 units and starts at an angle of $0.2 \times 2\pi$. Test particles are initially in an anticlockwise circular orbit around the central mass. (Perturbation direction set to 1)



(a) t = 10.0s, Scale = 30 x 30 square units.

(b) t = 20.0s, Scale = 30 x 30 square units.

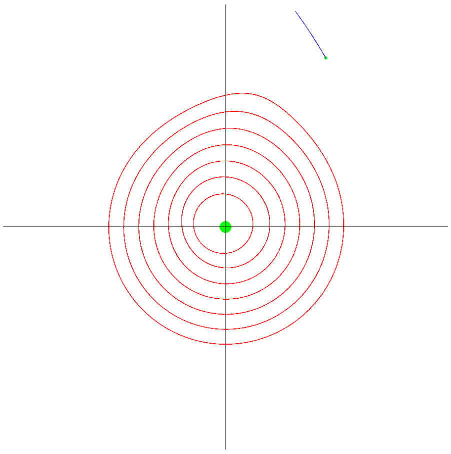
(c) t = 30.0s, Scale = 30 x 30 square units.

(d) t = 40.0s, Scale = 30 x 30 square units.

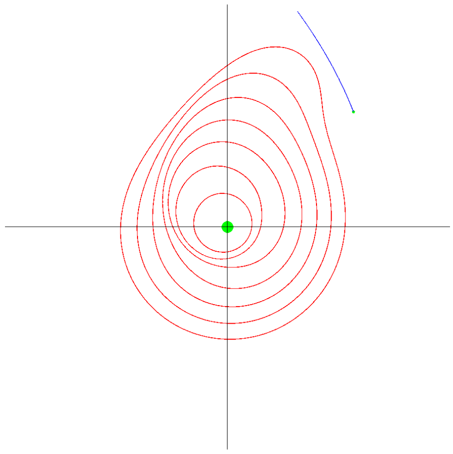
(e) t = 50.0s, Scale = 30 x 30 square units.

(f) t = 60.0s, Scale = 30 x 30 square units.

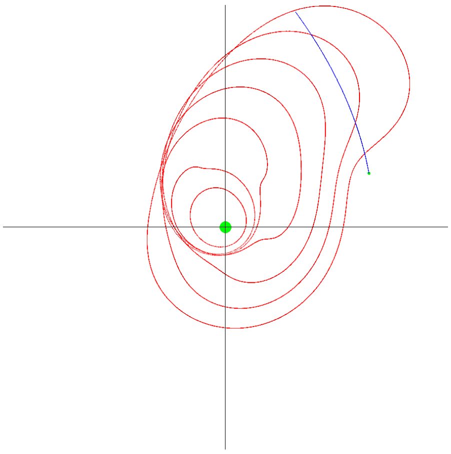
Figure 6: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 10 units and starts at an angle of $0.2 \times 2\pi$. Test particles are initially in a clockwise circular orbit around the central mass. (Perturbation direction set to 1)



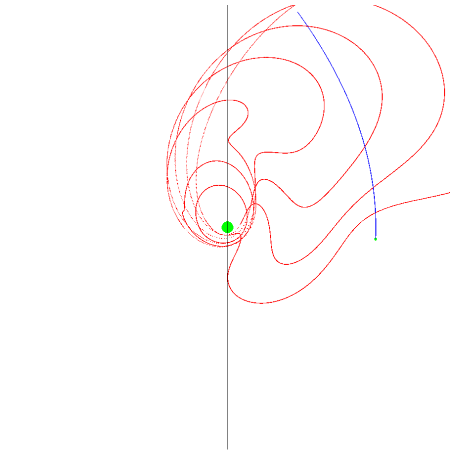
(a) t = 10.0s, Scale = 30 x 30 square units.



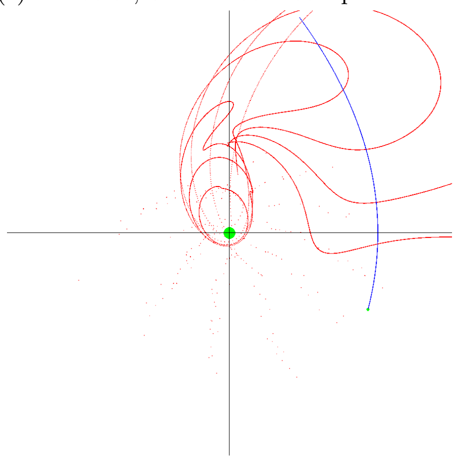
(b) t = 20.0s, Scale = 30 x 30 square units.



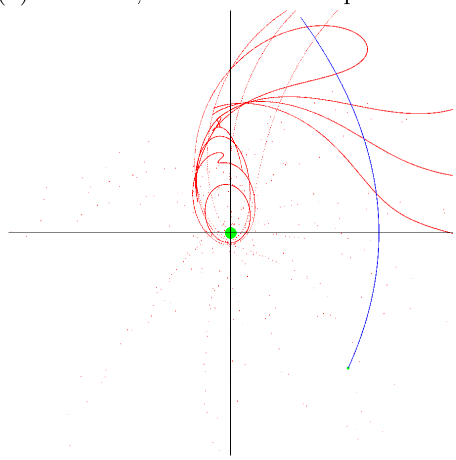
(c) t = 30.0s, Scale = 30 x 30 square units.



(d) t = 40.0s, Scale = 30 x 30 square units.

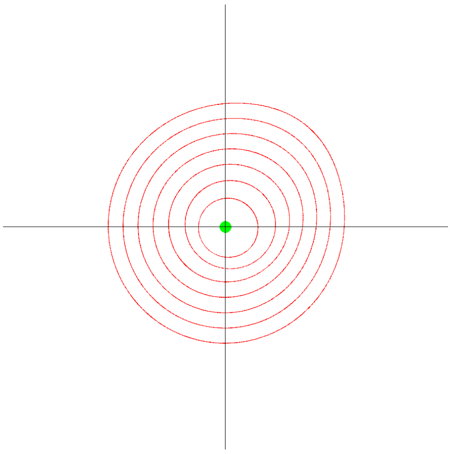


(e) t = 50.0s, Scale = 30 x 30 square units.

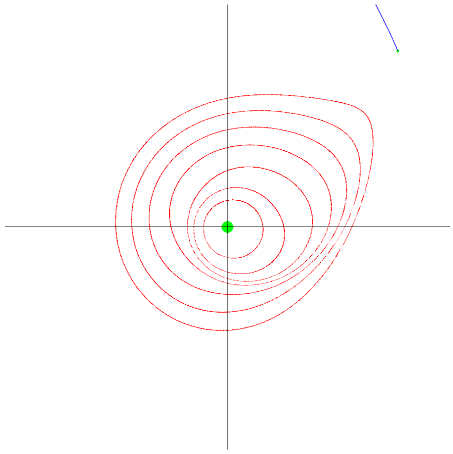


(f) t = 60.0s, Scale = 30 x 30 square units.

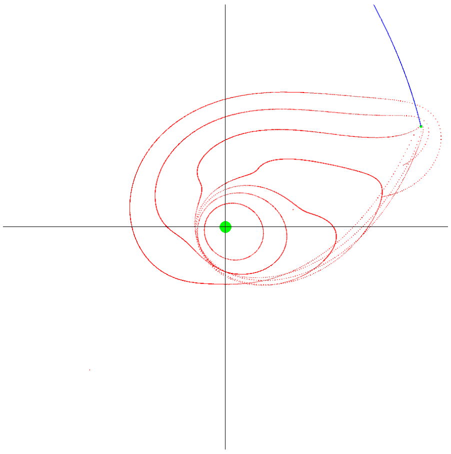
Figure 7: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 10 units and starts at an angle of $0.2 \times 2\pi$. Test particles are initially in an anticlockwise circular orbit around the central mass. (Perturbation direction set to 1)



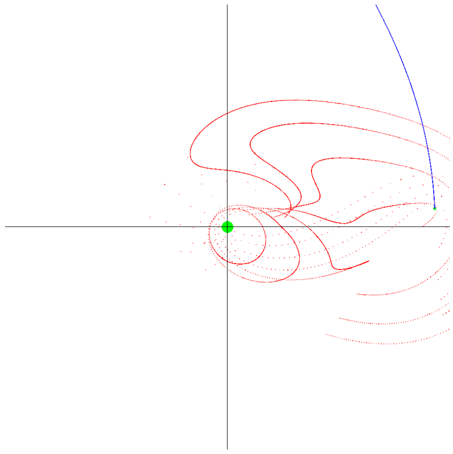
(a) t = 15.0s, Scale = 30 x 30 square units.



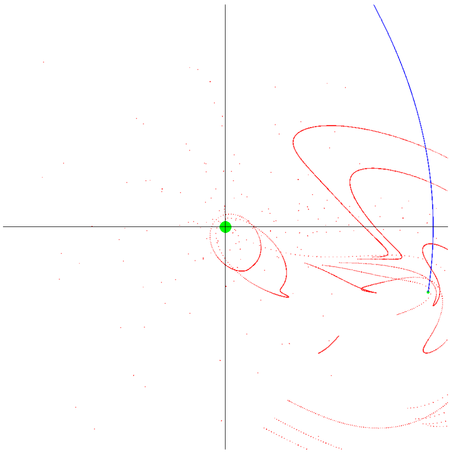
(b) t = 30.0s, Scale = 30 x 30 square units.



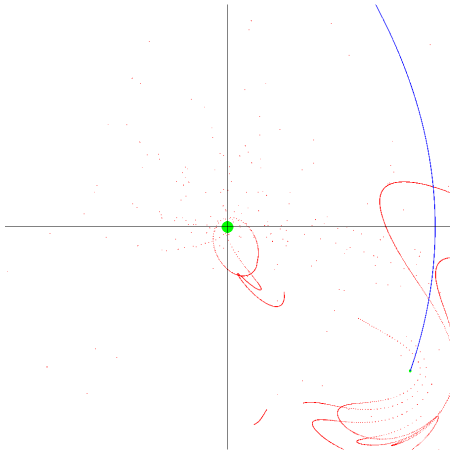
(c) t = 45.0s, Scale = 30 x 30 square units.



(d) t = 60.0s, Scale = 30 x 30 square units.

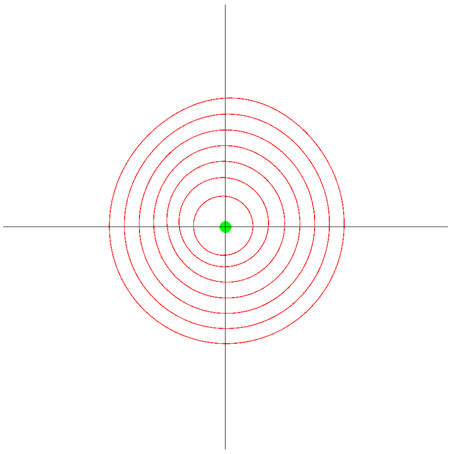


(e) t = 75.0s, Scale = 30 x 30 square units.

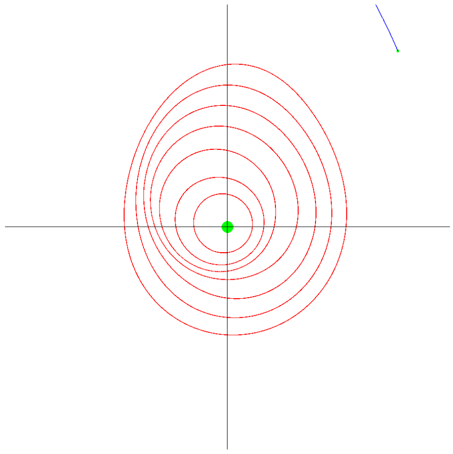


(f) t = 90.0s, Scale = 30 x 30 square units.

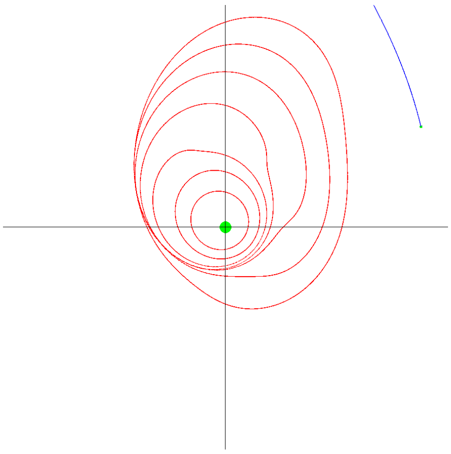
Figure 8: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 14 units and starts at an angle of $0.2 \times 2\pi$. Test particles are initially in a clockwise circular orbit around the central mass. (Perturbation direction set to 1)



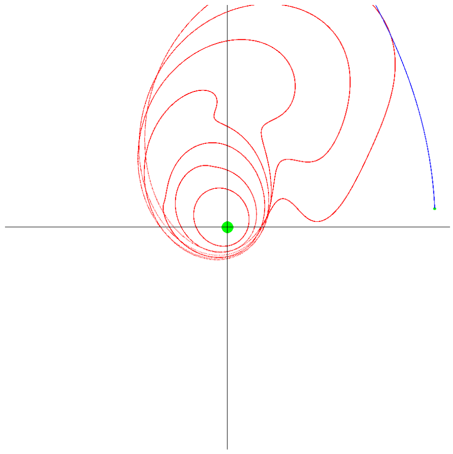
(a) t = 15.0s, Scale = 30 x 30 square units.



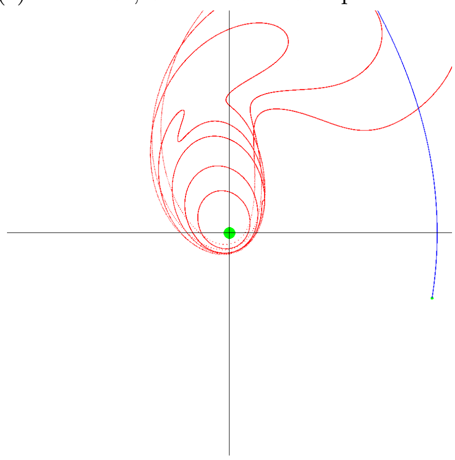
(b) t = 30.0s, Scale = 30 x 30 square units.



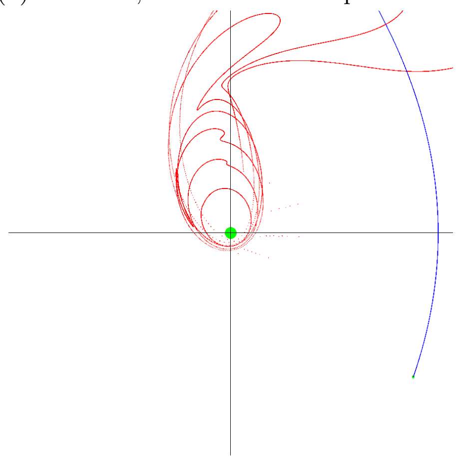
(c) t = 45.0s, Scale = 30 x 30 square units.



(d) t = 60.0s, Scale = 30 x 30 square units.

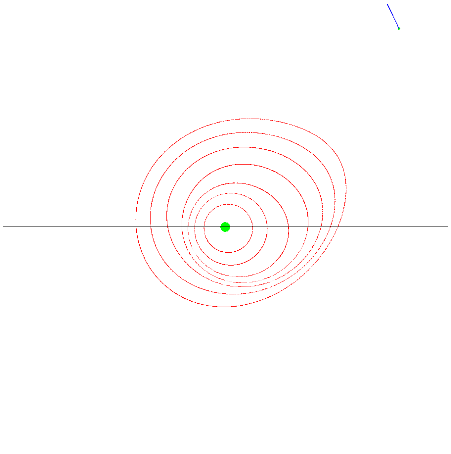


(e) t = 75.0s, Scale = 30 x 30 square units.

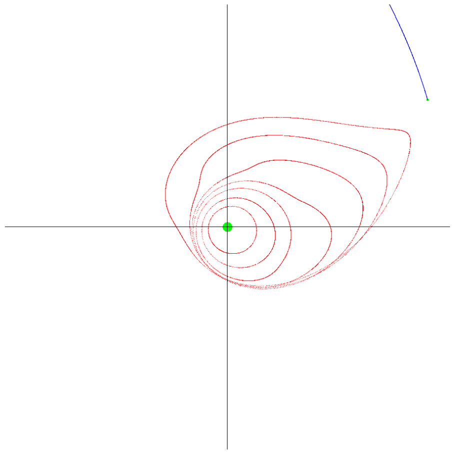


(f) t = 90.0s, Scale = 30 x 30 square units.

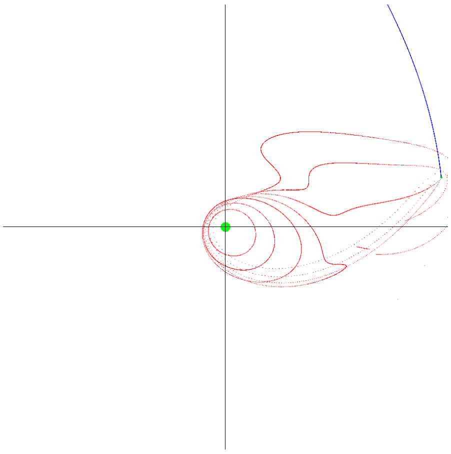
Figure 9: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 14 units and starts at an angle of $0.2 \times 2\pi$. Test particles are initially in an anticlockwise circular orbit around the central mass. (Perturbation direction set to 1)



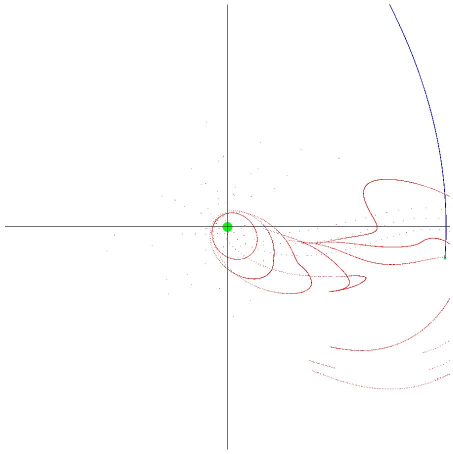
(a) t = 40.0s, Scale = 36.64 x 36.64 square units.



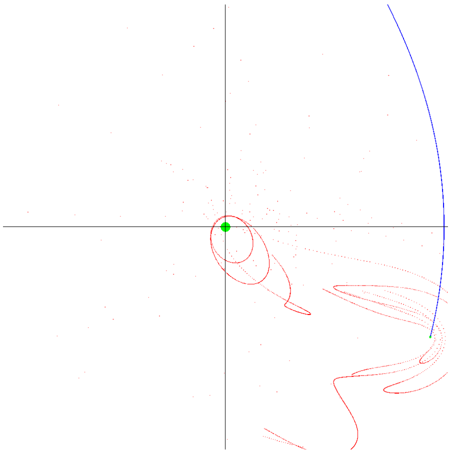
(b) t = 60.0s, Scale = 36.64 x 36.64 square units.



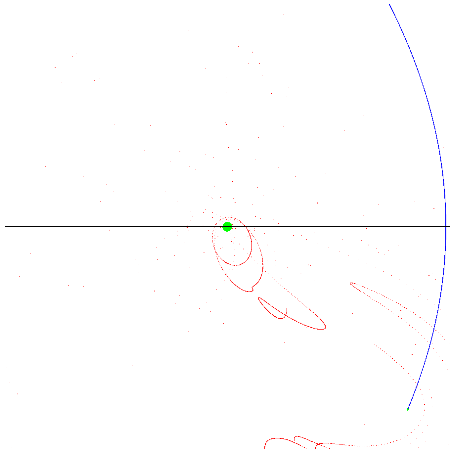
(c) t = 80.0s, Scale = 36.64 x 36.64 square units.



(d) t = 100.0s, Scale = 36.64 x 36.64 square units.

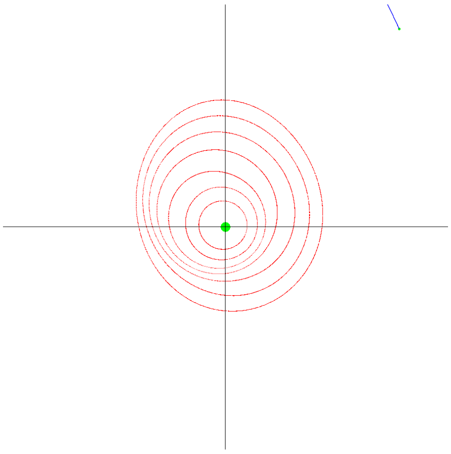


(e) t = 120.0s, Scale = 36.64 x 36.64 square units.

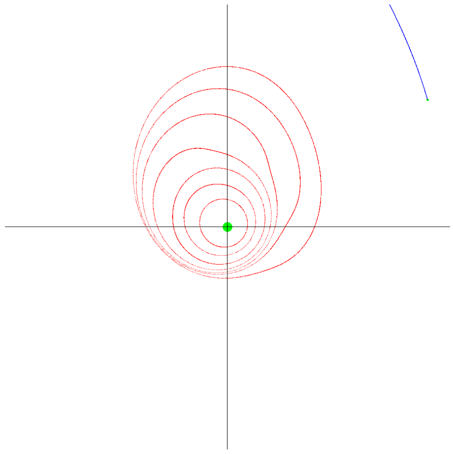


(f) t = 140.0s, Scale = 36.64 x 36.64 square units.

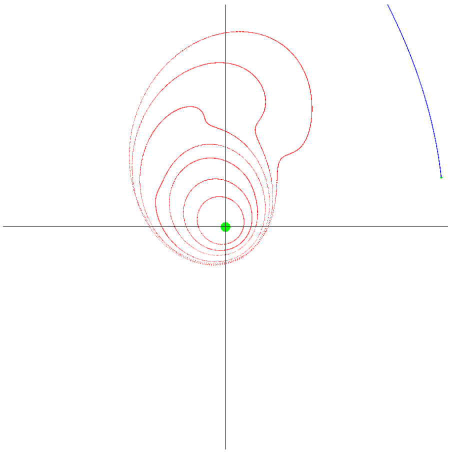
Figure 10: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 18 units and starts at an angle of $0.2 \times 2\pi$. Test particles are initially in a clockwise circular orbit around the central mass. (Perturbation direction set to 1)



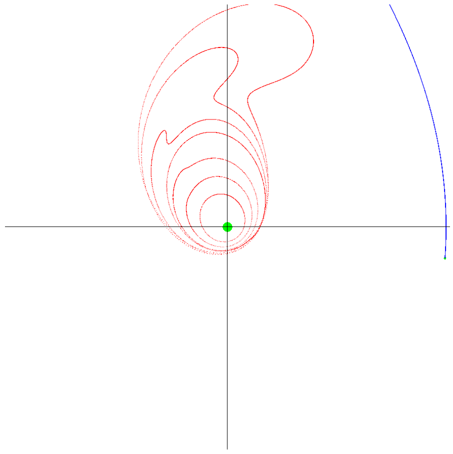
(a) t = 40.0s, Scale = 36.64 x 36.64 square units.



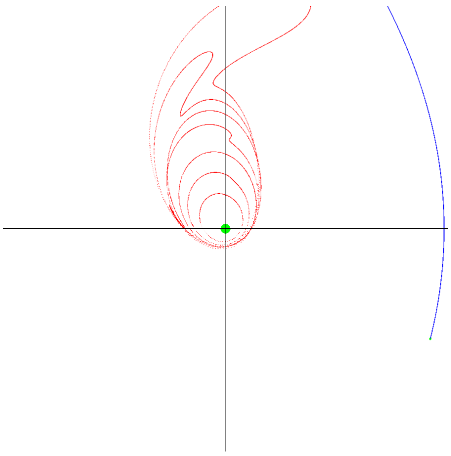
(b) t = 60.0s, Scale = 36.64 x 36.64 square units.



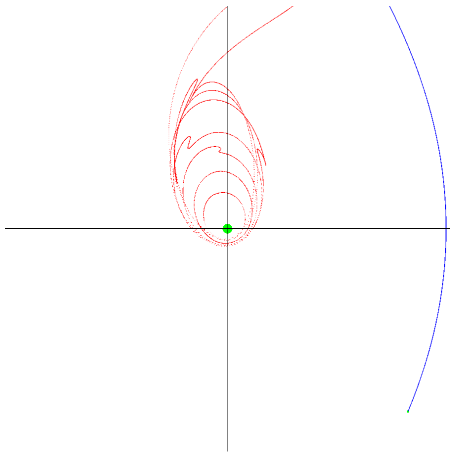
(c) t = 80.0s, Scale = 36.64 x 36.64 square units.



(d) t = 100.0s, Scale = 36.64 x 36.64 square units.



(e) t = 120.0s, Scale = 36.64 x 36.64 square units.



(f) t = 140.0s, Scale = 36.64 x 36.64 square units.

Figure 11: Plots at 6 different time slices for a perturbing galaxy with a parabolic orbit (e = 1). The perturbing galaxy has a closest approach of 18 units and starts at an angle of $0.2 \times 2\pi$. Test particles are initially in an anticlockwise circular orbit around the central mass. (Perturbation direction set to 1)

5 Conclusions

6 Instructions

6.1 Building

Software Required to Build Program:

1. C++ compiler (GNU g++)
2. SDL2
3. OpenGL
4. GLEW
5. cmake

Instructions to build and run project:

```
cd {project-directory}
cmake .
make
bin/main {optional-arguments}
```

Command line arguments if 5 arguments supplied (eccentricity, θ_0, Closest Approach, Rotation direction of central galaxy ($-1 = \circlearrowright, 1 = \circlearrowleft$), Perturbation orbit direction).

Command line arguments if 4 arguments supplied (eccentricity, θ_0, Closest Approach, TESTING (1 = true, 0 = false))

Command line arguments if 1 argument supplied (INTERACTIVE (1 = true, 0 = false))

Other combinations will result in a default simulation being carried out.

Logging Information: Each time step during logging is output to a new line in the outputted data file so that each line is of the format t,x1,y1,z1,x2,y2,z2,...,xN,yN,zN. A basic Python script could easily be created to carry out basic text processing on this data file to extract all of the particle's positions at a specific time. The extracted positions could then be written to a new text file in a plot friendly format i.e. x1,y2 newline x2,y2 etc. and then plotted using matplotlib all within the same script. This script was not created as it was felt that the screenshots provided sufficient graphical information.

6.2 Controls

- Pan using WASD ↑←↓→.
- Zoom out/in using QE $-+$.
- Take Screenshot using P.
- Start/Stop data logging to text file using L.
- Start/Pause/Unpause simulation using SPACEBAR.
- If (INTERACTIVE = true) left click, drag then release to create massive particle.

7 Code Listings

7.1 radius_plot.p

```
set terminal pngcairo size 900,900 enhanced font 'Verdana,10'
set output 'Radius.png'
set title "Radius of Circular orbit with time, Time Period T = 2 pi sqrt(r^3/(GM))"
set xlabel "Time (s)"
set ylabel "Radius (arbitrary units)"
set grid
set datafile separator ","
plot 'radius_data.txt' using 1:2 smooth bezier
```

7.2 convert_png.sh

```
#!/bin/bash
#Pass absolute or relative paths to tga files as arguments.
#Or use wildcards *.tga to specify all tga files in working directory.
for tga; do
    png="${tga%.tga}.png"
    echo converting "$tga"
    convert "$tga" "$png"
done
```

7.3 sdl_guard.h

```
#ifndef sdl_h
#define sdl_h
    #ifdef __APPLE__
        #include <SDL2/SDL.h>
    #elif __WIN32
        #include <SDL/SDL.h>
    #else
        #include <SDL2/SDL.h>
    #endif

#endif /* sdl_h */
```

7.4 main.cpp

```
// C++ Headers
#include <string>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
#include <sstream>

// OpenGL / glew / SDL Headers
#define GL3_PROTOTYPES 1
#include <GL/glew.h>
#include <SDL_opengl.h>


#include "utilities/sdl_guard.h"

// Headers
#include "physics/universe.h"
#include "physics/particle.h"
#include "capture/screenshot.h"
#include "capture/logger.h"
#include "utilities/utilities.h"
#include "utilities/camera.h"


std::string PROGRAMNAME = "Tidal Tails";
GLboolean AUTOSCREENSHOT = true;
GLint N = 40;
GLint WIDTH = 900;
GLint HEIGHT = 900;
// SDL
SDL_Window *mainWindow;
SDL_GLContext mainContext;

//initialises openGL and sdl
GLboolean init();
//checks SDL errors
void check_SDL_error(int line);
void run_simulation(var, var, var, GLint, GLint);
//closes sdl and openGL contexts
void cleanup();
//generates system with perturbing galaxy
void gen_perturbation(universe*, var e, var orbit_fraction, var closest_approach,
                      GLint central_rotation, GLint pert_direction, GLint N);
//generates single particle on conic orbit for testing purposes
void orbit_test(universe* u, var e, var orbit_fraction, var closest_approach);


int main(int argc, char *argv[]) {
    if (!init()) return -1;
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    SDL_GL_SwapWindow(mainWindow);
    if(argc==6) run_simulation(atof(argv[1]),atof(argv[2]),atof(argv[3]),
                               atoi(argv[4]),atoi(argv[5]));
```

```
    else if(argc==5 and atoi(argv[4])==1) {
        TESTING = true;
        run_simulation(atof(argv[1]),
                       atof(argv[2]), atof(argv[3]), 1, 1);
    }
    else if(argc==2 and atoi(argv[1])==1){
        INTERACTIVE = true;
        run_simulation(0,0,0,0,0);
    }
    else run_simulation(1.0,0.2,10.0,1,1);

    cleanup();
    return 0;
}

GLboolean init() {
    // Initialize SDL Video
    if (SDL_Init(SDL_INIT_VIDEO) < 0) {
        std::cout << "Failed to init SDL\n";
        return false;
    }

    mainWindow = SDL_CreateWindow(PROGRAMNAME.c_str(), SDL_WINDOWPOS_CENTERED,
                                  SDL_WINDOWPOS_CENTERED, WIDTH, HEIGHT,
                                  SDL_WINDOW_OPENGL);

    // SDL error check
    if (!mainWindow) {
        std::cout << "Unable to create window\n";
        check_SDL_error(__LINE__);
        return false;
    }

    // Create openGL context
    mainContext = SDL_GL_CreateContext(mainWindow);

    // Use GLCore
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_PROFILE_CORE);

    // Use OpenGL 3.2
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 2);
    SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);

    // Buffer swap synchronized with monitor's vertical refresh rate
    SDL_GL_SetSwapInterval(1);
    // Init GLEW macOS
#ifndef __APPLE__
    glewExperimental = GL_TRUE;
    glewInit();
#endif

    return true;
}

void gen_perturbation(universe* u, var e, var orbit_fraction, var closest_approach,
                      GLint central_rotation, GLint pert_direction, GLint N){
```

```
    var theta = 2.0*M_PI *(orbit_fraction);
    var rmin = closest_approach;
    var r = (1+e)*rmin/(1+e*cos(theta));
    var x0 = r * cos(theta);
    var y0 = r * sin (theta);
    var dvx, dvy;
    //fixes y0==0 error
    if(orbit_fraction == 0){
        dvx=0;
        dvy=pert_direction*1.0;
    }
    else if(orbit_fraction == 0.5){
        dvx = 0;
        dvy = -pert_direction*1.0;
    }
    else if(orbit_fraction == 0.25){
        dvx = pert_direction*1.0;
        dvy = 0.0;
    }
    else if(orbit_fraction == 0.75){
        dvx = -pert_direction*1.0;
        dvy = 0.0;
    }
    else {
        dvx = 1.0;
        dvy = -((1 - e * e) * x0 + e * (1 + e) * rmin) / y0;
    }
    var v0 = sqrt(2/sqrt(x0*x0+y0*y0)+(e-1)/rmin)/sqrt(dvx*dvx+dvy*dvy);
    u->generate_galaxy({pert_direction*x0,pert_direction*y0,0.0},
                       {pert_direction*v0*dvx,pert_direction*v0*dvy,0},
                       0.1,1.0,0.0,1,{{}},0);
    u->create_trail(u->particles.size()-1);
    u->generate_galaxy({0,0,0.0},{0,0,0},0.4,1.0,0.0,central_rotation,
                       {{N*12,2},{N*18,3},{N*24,4},{N*30,5},{N*36,6},{N*42,7},{N*48,8}},1);
}

void orbit_test(universe* u, var e, var orbit_fraction, var closest_approach){
    var theta = 2.0*M_PI *(orbit_fraction);
    var rmin = closest_approach;
    var r = (1+e)*rmin/(1+e*cos(theta));
    var x0 = r * cos(theta);
    var y0 = r * sin (theta);
    var dvx,dvy;
    //fixes y0==0 error.
    if(orbit_fraction == 0){
        dvx=0;
        dvy=1.0;
    }
    else if(orbit_fraction == 0.5){
        dvx = 0;
        dvy = -1.0;
    }
    else if(orbit_fraction == 0.25){
        dvx = 1.0;
        dvy = 0.0;
    }
    else if(orbit_fraction == 0.75){
```

```
        dvx = -1.0;
        dvy = 0.0;
    }
    else {
        dvx = 1.0;
        dvy = -((1 - e * e) * x0 + e * (1 + e) * rmin) / y0;
    }
    var v0 = sqrt(2/sqrt(x0*x0+y0*y0)+(e-1)/rmin)/sqrt(dvx*dvx+dvy*dvy);
    u->generate_galaxy({x0,y0,0.0},{v0*dvx,v0*dvy,0},0.4,0.0,0.0,1,{{}},0);
    u->create_trail(u->particles.size()-1);
    u->generate_galaxy({0,0,0.0},{0,0,0},0.4,1.0,0.0,1.0,{{}},1);

}

void run_simulation(var eccentricity, var orbit_fraction, var closest_approach,
                    GLint central_rotation, GLint pert_direction) {

    GLboolean mouseAllowed;
    std::fstream radius_Data;
    universe universe1 = universe(true);
    //Create perturbing galaxy
    if(!INTERACTIVE and !TESTING) {
        gen_perturbation(&universe1, eccentricity, orbit_fraction, closest_approach,
                         central_rotation, pert_direction, N);
        mouseAllowed = false;
    }
    else if(TESTING){
        radius_Data.open("radius_data.txt",std::fstream::out | std::fstream::trunc);
        orbit_test(&universe1,eccentricity,orbit_fraction,closest_approach);
        mouseAllowed = false;
    }
    else{
        mouseAllowed = true;
    }

    logger logger1 = logger();
    camera c = camera(WIDTH,HEIGHT);
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    universe1.render_universe(&c);
    SDL_GL_SwapWindow(mainWindow);
    GLboolean loop = true;
    GLboolean paused = true;
    GLboolean logging = false;
    GLboolean reversed = false;
    std::stringstream screenshot_title;

    var dx,dy,zl, mousex, mousey;
    GLdouble t = 0;
    dx = 0.4;
    dy = 0.4;
    zl = 0.1;
    mousex = 0;
    mousey = 0;

    while (loop == true) {
        SDL_Event event;
```

```
while (SDL_PollEvent(&event)) {
    if (event.type == SDL_QUIT) loop = false;
    if (event.type == SDL_MOUSEBUTTONDOWN and mouseAllowed) {
        switch (event.button.button){
            case SDL_BUTTON_LEFT:
                mousex = event.button.x;
                mousey = event.button.y;
                paused=true;
                break;
        }
    }
    if (event.type == SDL_MOUSEBUTTONUP and mouseAllowed) {
        switch (event.button.button){
            case SDL_BUTTON_LEFT:
                universe1.generate_galaxy(
                        {openGLpos(mousex,0,&c), openGLpos(mousey,1,&c),0},
                        {(15.0/(WIDTH))/c.zoom*(event.button.x-mousex),
                         (-15.0/(HEIGHT))/c.zoom*(event.button.y-mousey),
                         0.0},
                        0.4,1.0,0.0,1,{{}},0);
                universe1.create_trail(universe1.particles.size()-1);
                universe1.apply_first_step_single_particle();
                paused=false;
                break;
        }
    }

    if (event.type == SDL_KEYDOWN) {
        switch (event.key.keysym.sym) {
            case SDLK_ESCAPE:
                loop = false;
                break;
            case SDLK_p:
                screenshot_title.str(std::string());
                screenshot_title << "screenshot-";
                screenshot_title << std::fixed << std::setprecision(2)
                                 << t << "_("
                                 << c.position[0]-1.0*SCALE/c.zoom
                                 << "_" << c.position[0]+1.0*SCALE/c.zoom
                                 << ")("
                                 << c.position[1]-1.0*SCALE/c.zoom
                                 << "_" << c.position[1]+1.0*SCALE/c.zoom
                                 << ").tga";
                screenshot(screenshot_title.str());
                std::cout << "Screenshot created." << std::endl;
                break;
            case SDLK_SPACE:
                paused = not paused;
                if(t==0){
                    universe1.apply_first_step();
                    universe1.render_universe(&c);
                    t += getTimestep(&universe1);
                    if(logging) logger1.log_positions(t,universe1.particles);
                }
                std::cout << "Pause status: "
                          << static_cast<int>(paused)
                          << std::endl;
```

```
        std::cout << "(" << c.position[0]-1.0*SCALE/c.zoom << ","
                  << c.position[0]+1.0*SCALE/c.zoom << ")";
        std::cout << "(" << c.position[1]-1.0*SCALE/c.zoom << ","
                  << c.position[1]+1.0*SCALE/c.zoom << ")   t = "
                  << t << "\n";
        break;
    /*case SDLK_r:
        reversed= not reversed;
        //reverse time
        std::cout << "Time reversed." << std::endl;
        break;
     */
    case SDLK_l:
        logging = not logging;
        //start/stop logging
        if(logging){
            logger1.start("data-"+std::to_string(t)+".csv");
            logger1.log_positions(t,universe1.particles);
        }
        else logger1.stop();
        std::cout << "Logging status: "
                  << static_cast<int>(logging)
                  << std::endl;
        break;
    case SDLK_w:
        update_view(&c,0.0,dy/(c.zoom),0.0);
        if(paused){
            glClearColor(1.0, 1.0, 1.0, 1.0);
            glClear(GL_COLOR_BUFFER_BIT);
            universe1.render_universe(&c);
            SDL_GL_SwapWindow(mainWindow);
        }
        break;
    case SDLK_s:
        update_view(&c,0.0,-1.0*dy/(c.zoom),0.0);
        if(paused){
            glClearColor(1.0, 1.0, 1.0, 1.0);
            glClear(GL_COLOR_BUFFER_BIT);
            universe1.render_universe(&c);
            SDL_GL_SwapWindow(mainWindow);
        }
        break;
    case SDLK_a:
        update_view(&c,-1.0*dx/(c.zoom),0.0,0.0);
        if(paused){
            glClearColor(1.0, 1.0, 1.0, 1.0);
            glClear(GL_COLOR_BUFFER_BIT);
            universe1.render_universe(&c);
            SDL_GL_SwapWindow(mainWindow);
        }
        break;
    case SDLK_d:
        update_view(&c,dx/(c.zoom),0.0,0.0);
        if(paused){
            glClearColor(1.0, 1.0, 1.0, 1.0);
            glClear(GL_COLOR_BUFFER_BIT);
            universe1.render_universe(&c);
```

```
                    SDL_GL_SwapWindow(mainWindow);
                }
                break;
            case SDLK_q:
                update_view(&c,0.0,0.0,-1.0*zl);
                if(paused){
                    glClearColor(1.0, 1.0, 1.0, 1.0);
                    glClear(GL_COLOR_BUFFER_BIT);
                    universe1.render_universe(&c);
                    SDL_GL_SwapWindow(mainWindow);
                }
                break;
            case SDLK_e:
                update_view(&c,0.0,0.0,zl);
                if(paused){
                    glClearColor(1.0, 1.0, 1.0, 1.0);
                    glClear(GL_COLOR_BUFFER_BIT);
                    universe1.render_universe(&c);
                    SDL_GL_SwapWindow(mainWindow);
                }
                break;
            default:
                break;
        }
    }
}

if(!paused) {
    universe1.update(mainWindow,&c, reversed);
    t += getTimestep(&universe1);
    if(logging){
        logger1.log_positions(t,universe1.particles);
    }
    if(TESTING){
        radius_Data <<  t << ","
                    << std::sqrt(getPosition(universe1.particles[0])[0]*
                                 getPosition(universe1.particles[0])[0]+
                                 getPosition(universe1.particles[0])[1]*
                                 getPosition(universe1.particles[0])[1])
                    << "\n";
    }
    if(int(t)%5==0 and (t-int(t)) < getTimestep(&universe1) and AUTOSCREENSHOT){
        screenshot_title.str(std::string());
        screenshot_title << "screenshot-";
        screenshot_title << std::fixed << std::setprecision(2)
                         << t << "_("
                         << c.position[0]-1.0*SCALE/c.zoom
                         << "_" << c.position[0]+1.0*SCALE/c.zoom
                         << ")("
                         << c.position[1]-1.0*SCALE/c.zoom
                         << "_" << c.position[1]+1.0*SCALE/c.zoom
                         << ").tga";
        screenshot(screenshot_title.str());
        std::cout << "Screenshot created." << std::endl;
    }
}
```

```
    }
}

void cleanup() {
    SDL_GL_DeleteContext(mainContext);
    SDL_DestroyWindow(mainWindow);
    SDL_Quit();
}

void check_SDL_error(GLint line = -1) {
    std::string error = SDL_GetError();
    if (error != "") {
        std::cout << "SDL Error : " << error << std::endl;

        if (line != -1) std::cout << "\nLine : " << line << std::endl;

        SDL_ClearError();
    }
}
```

7.5 logger.h

```
#ifndef LOGGER_H
#define LOGGER_H
#include <fstream>
#include <vector>
#include "utilities/utilities.h"
#include "physics/particle.h"

class particle;
//logs data that can be later plotted with gnuplot
class logger{
private:
    std::fstream f;
    std::string title;
public:
    void log_positions(var t, std::vector<particle*> particles);
    logger();
    void stop();
    void start(std::string);

};

#endif //LOGGER_H
```

7.6 logger.cpp

```
#include "capture/logger.h"
#include <iostream>
logger::logger(){

}

void logger::start(std::string s){
    f.open(s,std::fstream::out | std::fstream::trunc);
    title = s;
    std::cout << title << " opened.\n";
}

void logger::stop(){
    f.close();
    std::cout << title << " closed.\n";
}

void logger::log_positions(var t, std::vector<particle *> particles) {
    f << t;
    for(int i = 0; i<particles.size(); i++){
        f << "," << to_string(getPosition(particles[i]));
    }
    f << '\n';
}
```

7.7 screenshot.h

```
#ifndef SCREENSHOT_H
#define SCREENSHOT_H
#include "utilities/sdl_guard.h"
#include <GL/glew.h>
#include <fstream>

void screenshot (std::string);

#endif //SCREENSHOT_H
```

7.8 screenshot.cpp

```
#include "capture/screenshot.h"
// Original Code credit http://www.flashbang.se/archives/155 (Heavily modified)

void screenshot (std::string filename){
    GLint size[4];
    glGetIntegerv(GL_VIEWPORT, size);
    glReadBuffer(GL_FRONT);
    GLint64 imageSize = size[2] * size[3] * 3;
    GLubyte *data = new GLubyte[imageSize];
    glReadPixels(0,0,size[2],size[3], GL_BGR,GL_UNSIGNED_BYTE,data);
    GLint x0= size[2] % 256;
    GLint x1= (size[2]-x0)/256;
    GLint y0= size[3] % 256;
    GLint y1= (size[3]-y0)/256;
    //.tga file format header
    GLubyte header[18]={0,0,2,0,0,0,0,0,0,0,0,0,
                        static_cast<GLubyte >(x0),
                        static_cast<GLubyte >(x1),
                        static_cast<GLubyte >(y0),
                        static_cast<GLubyte >(y1),24,0};

    std::fstream File(filename, std::ios::out | std::ios::binary);
    File.write (reinterpret_cast<char *>(header), sizeof(GLbyte)*18);
    File.write (reinterpret_cast<char *>(data), sizeof(GLbyte)*imageSize);
    File.close();

    delete[] data;
    data=NULL;
}
```

7.9 particle.h

```
#ifndef PARTICLE_H
#define PARTICLE_H
#include "utilities/sdl_guard.h"
#include "utilities/utilities.h"
#include "utilities/camera.h"
#include <GL/glew.h>
#include <array>

class camera;

class particle {
private:
    var radius;
    var mass;

    vec4 color;
    vec3 position;
    vec3 position_old;
    vec3 velocity;
    vec3 acceleration;
public:
    GLboolean isFixed;
    friend const var& getRadius(particle*);
    friend const var& getMass(particle*);
    friend const vec3& getPosition(particle*);
    friend const vec3& getPositionOld(particle*);
    friend const vec3& getVelocity(particle*);
    friend const vec3& getAcceleration(particle*);
    friend const vec4& getColor(particle*);
    friend void update_particle(particle* p,vec3 x,vec3 v, vec3 a);
    friend void update_particle_internal(particle* p, var R, var M);
    friend void render(camera*, particle*);
    particle(var m, var r, vec3 x0, vec3 v0, vec4 C, GLboolean fixed);
};


#endif //PARTICLE_H
```

7.10 particle.cpp

```
#include "physics/particle.h"

const var& getMass(particle* a){
    return a->mass;
}
const var& getRadius(particle* a){
    return a->radius;
}
const vec3& getPosition(particle* a){
    return a->position;
}

const vec3& getPositionOld(particle* a){
    return a->position_old;
}
```

```
const vec3& getVelocity(particle* a){
    return a->velocity;
}
const vec3& getAcceleration(particle* a){
    return a->acceleration;
}

const vec4& getColor(particle* a){
    return a->color;
}

particle::particle(var m, var r, vec3 x0, vec3 v0, vec4 C, GLboolean fixed) {
    mass = m;
    radius = r;
    position = x0;
    velocity = v0;
    acceleration = {0.0,0.0,0.0};
    color = C;
    isFixed = fixed;
}

void update_particle(particle* p,vec3 x,vec3 v, vec3 a){
    if(!p->isFixed){
        p->acceleration = a;
        p->velocity = v;
        p->position_old = p->position;
        p->position = x;
    }
}

void update_particle_internal(particle* p, var R, var M){
    p->mass = M;
    p->radius = R;
}

void render(camera*  c, particle* a){
    GLint subdivisions = 20;
    GLUquadricObj *quadric = gluNewQuadric();
    var color[4] = {getColor(a)[0],getColor(a)[1],getColor(a)[2],getColor(a)[3]};
    glColor4dv(color);
    gluQuadricNormals(quadric, GLU_SMOOTH);
    glPushMatrix();
    glTranslatef(c->zoom*(getPosition(a)[0]-c->position[0])/SCALE,
                 c->zoom*(getPosition(a)[1]-c->position[1])/SCALE,
                 c->zoom*(getPosition(a)[2]-c->position[2])/SCALE);
    gluSphere(quadric, c->zoom*getRadius(a)/SCALE, subdivisions,subdivisions);
    //glRotatef(0.01,0.0,0.0,1.0);
    glPopMatrix();
    gluDeleteQuadric(quadric);
}
```

7.11 universe.h

```
#ifndef UNIVERSE_H
#define UNIVERSE_H
#include <vector>
#include <array>
#include <cmath>
#include "utilities/utilities.h"
#include "physics/particle.h"
#include "capture/logger.h"
#include "utilities/camera.h"
class particle;
class camera;

class universe{
private:
    std::vector<GLint> galaxy_index;
    std::vector<GLint> trails_kept;
    std::vector<std::vector<vec3> > particle_trails;
    var time;
    var dt;
    var M_max;
    var M_min;
    var R_max;
    var R_min;

    var prev_time;
    var G;
    GLboolean particles_massless;
    //distribution is (density, radius)
    void apply_forces();

public:
    std::vector<particle*> particles;
    void apply_first_step();
    void apply_first_step_single_particle();
    friend vec3 gforce(vec3 a0, particle*, particle*, var);
    void compute_forces();
    void create_trail(GLint particle_num);
    void update(SDL_Window* mainWindow, camera* c, GLboolean isReversed);
    void generate_galaxy(vec3 x0,vec3 v0, var R, var mass, var mass_min, GLint rotation,
                         std::vector<std::array<GLint,2> > distribution, GLboolean fixed);
    void render_universe(camera* c);
    friend var getTimestep(universe*);
    universe(GLboolean);
};

#endif //UNIVERSE_H
```

7.12 universe.cpp

```
#include <iostream>
#include "physics/universe.h"
//sets initial parameters
universe::universe(GLboolean massless_particles){
    G = 1.0;
    M_max = 1.0;
    M_min = 0.00;
```

```
    R_max = 0.05;
    R_min = static_cast<var>(R_max/16.0);
    particles_massless = massless_particles;
    time = 0.0;
    dt = 0.005;

    galaxy_index.push_back(0);
    prev_time = 0.0;

}

void universe::render_universe(camera* c){
    //renders all large central masses
    for(GLint i = 0; i<galaxy_index.size()-1; i++){
        render(c,particles[galaxy_index[i]]);
    }
    //renders test masses
    GLboolean notLargeMass;
    for(GLint i = 0; i<particles.size();i++){
        notLargeMass=1;
        for(GLint j = 0; j<galaxy_index.size()-1; j++){
            if(i==galaxy_index[j]) notLargeMass=0;
        }
        if(notLargeMass) render(c,particles[i]);
    }
    //renders any trails
    for(int i=0;i<trails_kept.size();i++){
        for(int j=1;j<particle_trails[i].size(); j++){
            glLineWidth(1.5);
            glColor4f(0.0, 0.0, 1.0,1.0);
            glBegin(GL_LINES);
            glVertex3f(c->zoom/SCALE * (particle_trails[i][j-1][0]-c->position[0]),
                       c->zoom/SCALE * (particle_trails[i][j-1][1]-c->position[1]),
                       c->zoom/SCALE * (particle_trails[i][j-1][2]-c->position[2]));
            glVertex3f(c->zoom/SCALE * (particle_trails[i][j][0]-c->position[0]),
                       c->zoom/SCALE * (particle_trails[i][j][1]-c->position[1]),
                       c->zoom/SCALE * (particle_trails[i][j][2]-c->position[2]));
            glEnd();
        }
    }
    //renders grid lines
    render_grid(c);
}

void universe::generate_galaxy(vec3 x0 = {0.0,0.0,0.0},vec3 v0 = {0.0,0.0,0.0},
                               var R = 5.0, var mass = 1.0,
                               var mass_min = 0.0, GLint rotation = 1,
                               std::vector<std::array<GLint,2> > distribution ={{}},
                               GLboolean fixed = 0) {

    particles.push_back(new particle(mass,R,x0,v0,color_green,fixed));
    var theta = 0.0;
    vec3 x = {0.0,0.0,0.0};
    vec3 v = {0.0,0.0,0.0};
    var r_min = R/SCALE;
    var vscale = 0.0;
```

```
    //generates particle distribution
    for(GLint i = 0; i<distribution.size(); i++) {
        for (GLint j = 0; j < distribution[i][0]; j++) {
            vscale = static_cast<var >(sqrt(G * mass / (distribution[i][1])));
            theta = 2.0 * M_PI * j / distribution[i][0];
            v = {static_cast<var >(-rotation*vscale * sin(theta)),
                 static_cast<var >(rotation*vscale * cos(theta)), 0.0};
            x = {static_cast<var >(distribution[i][1] * cos(theta)),
                 static_cast<var >(distribution[i][1] * sin(theta)), 0.0};
            particles.push_back(new particle(mass_min, r_min, add(x, x0), add(v, v0),
                                             color_red ,0));
        }
    }
    galaxy_index.push_back(particles.size());
}

void universe::create_trail(GLint particle_num){
    if(!particles[particle_num]->isFixed) {
        trails_kept.push_back(particle_num);
        particle_trails.push_back({getPosition(particles[particle_num])});
    }
}

//updates system and renders result  (time steps by dt)
void universe::update(SDL_Window* mainWindow, camera* c, GLboolean isReversed) {
    if (isReversed) dt = -std::abs(dt);
    else dt = std::abs(dt);
    apply_forces();

    var current_time = SDL_GetTicks();
    //adaptive fps to render. 1000*100/N = FPS.
    //lim to 30
    var time_step = particles.size()/100.0;
    if(1000.0/time_step > 60.0 and !INTERACTIVE) time_step = 1000.0/60.0;
    if (current_time - prev_time > time_step){
        glClearColor(1.0, 1.0, 1.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
        render_universe(c);
        SDL_GL_SwapWindow(mainWindow);
        prev_time=current_time;
        for(int i=0;i<trails_kept.size();i++){
            particle_trails[i].push_back(getPosition(particles[i]));
        }
    }

    time+=dt;
}

//calculates force between particle p and b
vec3 gforce(vec3 a0,particle* p, particle* b, var G = 1.0){
    vec3 a;
    var R;
    R = dist(getPosition(p), getPosition(b));
    if (R > getRadius(b)+getRadius(p))
        a = add(a0, mul(-G * getMass(b) / std::pow((R), 2),
                        unit(getPosition(p), getPosition(b))));
    else// Is now repulsive force, could be linear force for inside sphere.
```

```
        a = add(a0, mul((-1.0)*-G * getMass(b) / (R*R) / std::pow(getRadius(b), 3),
                        unit(getPosition(p), getPosition(b))));
    return a;
}

//finds x0 v0 so verlet can start.
void universe::apply_first_step(){
    vec3 a;
    vec3 v;
    vec3 x;

    for(GLint i = 0; i < particles.size(); i++){
        a={0.0,0.0,0.0};
        if(particles_massless){
            for(GLint j = 0; j<galaxy_index.size()-1;j++){
                if(i==galaxy_index[j]) continue;
                else{
                    a = gforce(a, particles[i],particles[galaxy_index[j]],G);
                }
            }
        }
        else {
            for (GLint j = 0; j < particles.size(); j++) {
                if (i == j or getMass(particles[j]) < 0.000001) continue;
                else {
                    a = gforce(a,particles[i],particles[j],G);
                }
            }
        }
        v = add(getVelocity(particles[i]),
                mul(0.5*dt,add(a,getAcceleration(particles[i]))));
        x = add(add(getPosition(particles[i]),mul(dt,getVelocity(particles[i]))),
                mul(0.5*dt*dt,getAcceleration(particles[i])));
        update_particle(particles[i],x,v,a);
    }
    time+=dt;
}

//finds x0 v0 so verlet can start for particles from clicks
void universe::apply_first_step_single_particle(){
    vec3 a;
    vec3 v;
    vec3 x;

    GLint i = particles.size()-1;
    a={0.0,0.0,0.0};
    if(particles_massless){
        for(GLint j = 0; j<galaxy_index.size()-1;j++){
            if(i==galaxy_index[j]) continue;
            else{
                a = gforce(a, particles[i],particles[galaxy_index[j]],G);
            }
        }
    }
    else {
        for (GLint j = 0; j < particles.size(); j++) {
            if (i == j or getMass(particles[j]) < 0.000001) continue;
```

```
            else {
                a = gforce(a,particles[i],particles[j],G);
            }
        }
    }
    v = add(getVelocity(particles[i]),mul(0.5*dt,add(a,getAcceleration(particles[i]))));
    x = add(add(getPosition(particles[i]),mul(dt,getVelocity(particles[i]))),
            mul(0.5*dt*dt,getAcceleration(particles[i])));
    update_particle(particles[i],x,v,a);
}

//updates positions of particles according to verlet
void universe::apply_forces(){
    vec3 a;
    vec3 v;
    vec3 x;

    for(GLint i = 0; i < particles.size(); i++){
        a={0.0,0.0,0.0};
        if(particles_massless){
            for(GLint j = 0; j<galaxy_index.size()-1;j++){
                if(i==galaxy_index[j]) continue;
                else{
                    a = gforce(a, particles[i],particles[galaxy_index[j]],G);
                }
            }
        }
        else {
            for (GLint j = 0; j < particles.size(); j++) {
                if (i == j or getMass(particles[j]) < 0.000001) continue;
                else {
                    a = gforce(a,particles[i],particles[j],G);
                }
            }
        }
        //leapfrog
        //v=add(getVelocity(particles[i]),mul(dt,a));
        //x=add(getPosition(particles[i]),mul(dt,v));

        //verlet O(dt^4)
        x = add(add(mul(2.0,getPosition(particles[i])),
                    mul(-1.0,getPositionOld(particles[i]))),mul(dt*dt,a));
        //is one time step behind O(dt^2)
        v =  mul(1.0/(2.0*dt),add(x,mul(-1.0,getPositionOld(particles[i]))));
        update_particle(particles[i],x,v,a);
    }
}

var getTimestep(universe* a){
    return a->dt;
}
```

7.13 camera.h

```
#ifndef CAMERA_H
#define CAMERA_H
#include "physics/universe.h"
#include "physics/particle.h"
//camera for openGL context

class camera{
private:
    var zoom_level;

public:
    vec3 position;
    var zoom;
    GLint width,height;
    camera(GLint,GLint);
    friend void update_view(camera* c,var dx, var dy , var zl);
};


#endif //CAMERA_H
```

7.14 camera.cpp

```
#include "utilities/camera.h"

camera::camera(GLint w, GLint h){
    position = {0.0,0.0,0.0};
    zoom_level = 0.0;
    zoom = 1.0;
    width=w;
    height=h;
}

void update_view(camera* c,var dx, var dy, var zl) {
    c->position = {c->position[0]+dx,c->position[1]+dy,0.0};
    c->zoom_level += zl;
    c->zoom = std::exp(c->zoom_level);
}
```

7.15 utilities.h

```
#ifndef UTILITIES_H
#define UTILITIES_H

#include "sdl_guard.h"
#include <GL/glew.h>
#include <string>
#include <array>

typedef GLdouble var;
typedef std::array<var, 3> vec3;
typedef std::array<var, 4> vec4;
std::string format_time(GLdouble);

//Hard coded for 3D for speed
vec3 add(vec3,vec3);
vec3 sub(vec3,vec3);
vec3 mul(var,vec3);
vec3 mul(vec3, var);
var dot(vec3,vec3);
var abs(vec3);

var dist(vec3,vec3);
vec3 unit(vec3,vec3);
vec3 cross(vec3, vec3);

std::string to_string(vec3);
std::string to_string(vec4);

class camera;
#include "utilities/camera.h"
extern var SCALE;
extern var FPS;
extern GLboolean INTERACTIVE;
extern GLboolean TESTING;
var openGLpos(GLint x, GLboolean isy, camera* c);

void render_grid(camera* c);

//colors
extern vec4 color_red;
extern vec4 color_yellow;
extern vec4 color_green;
extern vec4 color_cyan;
extern vec4 color_blue;
extern vec4 color_magenta;
extern vec4 color_black;
extern vec4 color_white;

extern std::array<vec4*,6> color_list;

void render_sphere(camera* c, vec3 x, var R);

#endif //UTILITIES_H
```

7.16 utilities.cpp

```
#include "utilities/utilities.h"

std::string format_time(GLdouble t){
    GLint h = floor(t/3600);
    GLint m = floor(t/60-h*60);
    GLint s = round(t-m*60-h*3600);
    return std::to_string(h)+"."+std::to_string(m)+"."+std::to_string(s);
}

vec3 add(vec3 a,vec3 b){
    return vec3{{a[0]+b[0],a[1]+b[1],a[2]+b[2]}};
}
vec3 sub(vec3 a,vec3 b){
    return vec3{{a[0]-b[0],a[1]-b[1],a[2]-b[2]}};
}
vec3 mul(var a,vec3 b){
    return vec3{{a*b[0],a*b[1],a*b[2]}};
}
vec3 mul(vec3 a, var b){
    return vec3{{b*a[0],b*a[1],b*a[2]}};
}
var dot(vec3 a,vec3 b){
    return (a[0]*b[0]+a[1]*b[1]+a[2]*b[2]);
}
vec3 cross(vec3 a, vec3 b){
    return vec3{{a[1]*b[2]-a[2]*b[1],a[2]*b[0]-a[0]*b[2],a[0]*b[1]-a[1]*b[0]}};
}

var abs(std::array<var, 3> a){
    return static_cast<var>(sqrt(a[0]*a[0]+a[1]*a[1]+a[2]*a[2]));
}

var dist(vec3 a,vec3 b){
    return abs(sub(a,b));
}

vec3 unit(vec3 a,vec3 b){
    std::array<var,3> vec = sub(a,b);
    return mul(1.0/abs(vec),vec);
}

std::string to_string(vec3 a){
        return "{" + std::to_string(a[0]) + "," + std::to_string(a[1]) +
                "," + std::to_string(a[2])+"}";
}

std::string to_string(vec4 a){
    return "{" + std::to_string(a[0]) + "," + std::to_string(a[1]) +
            "," + std::to_string(a[2]) + "," + std::to_string(a[3]) + "}";
}

var SCALE = 15.0;

var FPS = 10.0;

GLboolean INTERACTIVE = false;
GLboolean TESTING = false;
```

```
//converts from pixel values to opengl context location.
var openGLpos(GLint x, GLboolean isy, camera* c){
    if(isy) return (((1.0 - 2.0*x/c->height))/c->zoom)*SCALE+c->position[1];
    else return (((2.0*x/c->width-1.0))/c->zoom)*SCALE+c->position[0];
}

vec4 color_red = {1.0,0.0,0.0,1.0};
vec4 color_yellow = {1.0,1.0,0.0,1.0};
vec4 color_green = {0.0,1.0,0.0,1.0};
vec4 color_cyan = {0.0,1.0,1.0,1.0};
vec4 color_blue = {0.0,0.0,1.0,1.0};
vec4 color_magenta= {1.0,0.0,1.0,1.0};
vec4 color_black = {0.0,0.0,0.0,1.0};
vec4 color_white = {1.0,1.0,1.0,1.0};

std::array<vec4*,6> color_list = {&color_red,&color_green,&color_blue,
                                  &color_yellow,&color_cyan,&color_magenta};


void render_sphere(camera* c, vec3 x, var R){
    GLint subdivisions = 20;
    GLUquadricObj *quadric = gluNewQuadric();
    glColor4d(0.0,0.0,1.0,1.0);
    gluQuadricNormals(quadric, GLU_SMOOTH);
    glPushMatrix();
    glTranslatef(c->zoom*(x[0]-c->position[0])/SCALE,
                 c->zoom*(x[1]-c->position[1])/SCALE,
                 c->zoom*(x[2]-c->position[2])/SCALE);
    gluSphere(quadric, R/(c->zoom*SCALE), subdivisions,subdivisions);
    //glRotatef(0.01,0.0,0.0,1.0);
    glPopMatrix();
    gluDeleteQuadric(quadric);
}

void render_grid(camera* c){
    glLineWidth(1.0);
    glColor4f(0.0, 0.0, 0.0,1.0);
    glBegin(GL_LINES);
    glVertex3f(-1.0,-(c->position[1]/SCALE)*c->zoom, 0.0);
    glVertex3f(1.0, -(c->position[1]/SCALE)*c->zoom, 0);
    glEnd();

    glLineWidth(1.0);
    glColor4f(0.0, 0.0, 0.0,1.0);
    glBegin(GL_LINES);
    glVertex3f(-(c->position[0]/SCALE)*c->zoom,-1.0, 0.0);
    glVertex3f(-(c->position[0]/SCALE)*c->zoom, 1.0, 0);
    glEnd();
}
//
```