Fireworks Simulation

Chip Bell, Timothy Herold, Charles Mood, Yogi Patel, Calvin Williams

---

1. INTRODUCTION

Through the course of this semester we have written a user application that simulates a fireworks display. The user can specify multiple launch parameters in order to modify the physically realistic launch trajectory, shell explosion, and fragment dispersion which will be simulated for the user by our program. The manner in which the fireworks fragments explode is guided to preserve physical accuracy. However, fragment particulars are randomized within a specified range such that each firework is unique and yet still follows realistic physical limits. The program can launch multiple fireworks, reports pertinent data to the user, and allows the user some control over the simulation by manipulating viewing position inside of the simulated scene using mouse and keyboard controls.

2. MATHEMATICAL MODEL

In order to visualize the motion of fireworks in real time, we must first have a method of calculating the positions of both the fireworks' shells and the shells' respective exploded fragments. This system can most easily be modeled by the standard equations for projectile motion. The standard model given in most introductory physics and calculus textbooks is something similar to the following [Smith and Minton 2006]:

$$\frac{d^2y}{dt^2} = -g$$

where g is the acceleration due to gravity. In this case, x is uninteresting because its acceleration is 0, and therefore has constant velocity. However, this model is unrealistic, because it does not consider the effects of air resistance. The force of drag in a fluid, in our case air, changes the way in which a projectile moves through it, changing its path based upon a drag coefficient. This leads us to form a more complex model that incorporates drag. The force resulting from drag F_d is given by the following equation [Fishbane et al. 2005]:

$$F_d = \frac{1}{2}\rho A C_d v^2$$

where ρ is the fluid density, A is the maximum cross-sectional area of the object, C_d is the drag coefficient of the moving object, and v the velocity of the moving object. Since we have the acting force, we can then solve for acceleration a by Newton's second law:

$$a = \frac{\rho A C_d v^2}{2m}$$

where m is the mass of the moving object. If we let

$$c = \frac{\rho A C_d}{2m}$$

we can then rewrite our equation as

$$a = cv^2$$

In our case, a and v are vectors because we are moving in space. So, we can rewrite the equation as a vector:

$$\begin{pmatrix} a_x \\ a_y \end{pmatrix} = c||\vec{v}|| \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

or

$$\begin{pmatrix} a_x \\ a_y \end{pmatrix} = c\sqrt{v_x^2 + v_y^2} \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

This new notation may not be intuitive at first. However, consider the magnitude of the velocity vector $\begin{pmatrix} v_x \\ v_y \end{pmatrix}$. Now that it has been multiplied by its own magnitude, the term $||\vec{v}|| \begin{pmatrix} v_x \\ v_y \end{pmatrix}$ has a magnitude $||\vec{v}||^2$.

Since acceleration and velocity are second and first derivatives of position, we can write this second-order system as a first-order system of ordinary differential equations. The below model incorporates both drag and gravity.

$$\frac{d^2x}{dt^2} = -c\frac{dx}{dt}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}$$

$$\frac{d^2y}{dt^2} = -c\frac{dy}{dt}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2} - g$$

This equation cannot be solved analytically, unlike its simpler counterpart, so a numerical approximation technique must be used instead to calculate the positional values of a projectile at a given time. Moreover, the equation is second order and many approximation techniques will not be applicable here. However, we can transform the equations in the following manner to create a first order system:

$$\frac{dx}{dt} = p$$

$$\frac{dy}{dt} = q$$

$$\frac{dp}{dt} = -cp\sqrt{p^2 + q^2}$$

$$\frac{dp}{dt} = -cq\sqrt{p^2 + q^2} - g$$

Now that we have a first-order system of ordinary differential equations, we can then apply an approximation method to attain (x, y) pairs for any given start conditions (x_0, y_0) and $(v_{x_0}, v_{y_0}) = (p_0, q_0)$.

In our case, we will apply a fourth-order Runge-Kutta method. This method was chosen over others because of its relative simplicity and small amount error compared to other methods, such as Euler's method [Burden and Faires 2005].

The system described above only propagates in two dimensions rather than three as would a real-life fireworks system. This problem can be resolved by storing the system in cylindrical coordinates. We assume no other forces are acting upon the firework at a given time other than gravity in the $-z$ direction and air-resistance working in both the vertical and horizontal axes of movement. So, we know that the object will only move within the plane in which it is launched and never deviate [Fishbane et al. 2005], meaning we can treat the system as a two-dimensional system rotated around the z-axis by some constant θ, i.e. cylindrical coordinates. Using our new representation, our system then becomes

$$\frac{dr}{dt} = p$$

$$\frac{dz}{dt} = q$$

$$\frac{dp}{dt} = -cp\sqrt{p^2 + q^2}$$

$$\frac{dp}{dt} = -cq\sqrt{p^2 + q^2} - g$$

Each shell is given its own initial radial angle θ, initial velocity v_0, and a launch azimuthal angle ϕ_0. Using the latter two, we can derive initial velocities both in the r and z directions. Lastly, we assume that shells are launched from the origin. Since, we now have all all initial conditions:

$$r_0 = 0, z_0 = 0,$$

$$\frac{dr_0}{dt} = p_0 = v_0 cos(\phi_0)$$

$$\frac{dz_0}{dt} = q_0 = v_0 sin(\phi_0)$$

we can step through the system using Runge-Kutta, and simulate the fireworks position starting from the time $t = 0$.

3. DATA MANAGEMENT

Although we have a model that is relatively accurate, we still have to be able to store pertinent data in the most cost effective way, and provide an interface for accessing this data, in order to visualize the data later in *OpenGL*.

This was acheived with a series of structs. The first two are "base types," `Vector2D` and `Vector3D`, holding points in two-dimensional and three-dimensional spaces respectvely. The next two structs, `Shell` and `Fragment` hold data pertinent to a fireworks shell and shell fragment in flight, such as position and mass. Given the number of shells to fire, an array is dynamically created to hold all of the shells

at one time at the beginning of execution. For each of those shells, an array of fragments is created, each with some percentage of their parent shell's mass. The total sum of the mass of all of these particles is always equals the mass of their parent shell, because we consider mass to be conserved in the explosion (although mass is not conserved afterwards).

Since the shells and fragments are propagating in cylindrical space, both `Fragment` and `Shell` have a `Vector2D` and a θ value associated with them. The cylindrical coordinates associated with a moving particle, be it a shell or fragment, is relative to its own point of origin. So, in the case of the shell, the point of origin is $(0, 0, 0)$. In order to find the Cartesian coordinates for a shell, we can simply use the basic conversion method for a cylindrical coordinate (r_c, θ_c, z_c) to a Cartesian coordinate (x_r, y_r, z_r) [Smith and Minton 2006]:

$$x_r = r_c cos(\theta_c)$$

$$y_r = r_c sin(\theta_c)$$

$$z_r = z_c$$

For fragments, however, the conversion is different. Since the fragments' point of origin is the point where they exploded, their position is relative to the shell. So, to convert to Cartesian, the coordinates must be converted separately and then added as vectors, yielding the following conversion for a fragment with position (r_f, θ_f, z_f) and explosion point (r_s, θ_s, z_s):

$$x = r_f cos(\theta_f) + r_s cos(\theta_s)$$

$$x = r_f sin(\theta_f) + r_s sin(\theta_s)$$

$$z = z_f + z_s$$

4. PHYSICAL ANALYSIS

As a result of the physically realistic model used for the propagation of the fireworks and their fragments, it is possible to monitor physical parameters assigned to the shells and fragments in order to extract information concerning the behavior of real world fireworks. Many different properties can be considered in this way; however, for this simulation, only those relevant to position and mass were chosen. Since these properties were already known due to their necessity for the propagation of the model, they were a natural choice. The center of mass and the bounding box were chosen to be monitored throughout the simulation.

4.1 Bounding Box Calculations

To better understand the behavior of the fragments produced by the explosion of the shells, we can consider the volume which encloses them throughout the simulation. This volume is given by the box just big enough to hold all of the fragments throughout the entire evolution of the fireworks display. Its sides are defined by the maximum and minimum values of fragment positions in the x, y, and z dimensions. Since these positions are updated at every step of the simulation

in time, the bounding box grows in real time as more fireworks explode. The bounding box provides information concerning the scale of the fireworks display in the sky as well as the maximum and minimum height of burning fragments. These factors are important for designing an impressive fireworks display as well as one which would be safe for viewers. Everything outside of the bounding box is guaranteed not to be in danger of coming into contact with burning firework material.

In the graphical display, the bounding box is shown by a white wireframe box. Should a burning fragment touch the ground, the color of the bounding box will change to red to represent danger. The bounding box display is toggled on and off with the B key.

4.2 Center of Mass Calculations

To better understand the motion of the fireworks as a whole, their center of mass was calculated throughout the simulation. This is the point in 3D space which behaves as if all of the mass of the shells and fragments were concentrated there and as if all forces acting on the shells and fragments acted upon that location. The center of mass responds to the force of the launching of each shell, the drag on the shells and fragments, and gravity. Tracking its location reveals the nature of the motion of the entire display. The calculation of the center of mass was performed using its definition as provided in [Halliday et al. 2005]:

$$x_{com} = \frac{1}{M}\sum_{i=1}^{n} m_i x_i$$

$$y_{com} = \frac{1}{M}\sum_{i=1}^{n} m_i y_i$$

$$z_{com} = \frac{1}{M}\sum_{i=1}^{n} m_i z_i$$

Each of the *x, y,* and *z* components are calculated separately, but all follow the same form. Given n total shells and fragments of combined mass M, the product of each shell or fragments mass and *x, y,* or *z* position is summed for every shell and fragment. The total is then divided by the total mass M. The 3D location of the center of mass is then given by this calculation for *x, y,* and *z*.

Two different forms of the center of mass calculation were considered. The first method considered all fireworks and fragments in the calculation, even those which had not yet been launched. This resulted in a center of mass which hung low to the ground at the start and slowly rose as the simulation progressed. The tendency of the center of mass to keep lower is a result of the number of shells waiting to be launched outweighing the shells and fragments in the air. While this is a true representation of the center of mass of the entire system, it provides little information about the dynamics in the sky at any given time. The second method considered for the center of mass calculations took into account only shells which had been launched into the air and all the burning fragments, which will be collectively referred to as the active mass. This method provides a constant source

of information concerning the motion of the fireworks display as seen by someone enjoying the show. The observer will generally be concerned with the active mass and not the large pile of fireworks waiting for launch, so this method of considering the center of mass is more relevant to the experience of someone at a fireworks show. The second method was used for our calculations as a result of its more intuitive nature.

In the graphical display, the location of the center of mass is shown by a white wireframe sphere. The radius of the sphere is a ratio of the active mass to the total mass of the fireworks display, so as the active mass gets larger and smaller, so too does the radius of the sphere. In this way, one can get a sense of the active mass as time progresses. The center of mass display is toggled on and off with the C key.

4.3 Data Window and Textual Readout

The data window provides a 2D display of the trajectory of the center of mass from a top down view. The trajectory is projected down into the x-y plane for display. Due to the ability of the client to make the show as long as desired, an issue arose with the amount of memory needed to save the entire trajectory to continue to have it display throughout the simulation. The amount of memory in this situation is unbounded. To resolve this issue, a matrix was allocated to represent the pixel grid of the data window. The following functions were used to map the x-y coordinates of the center of mass to the pixel grid of the data window as well as the matrix:

$$p_x = \frac{x_{com} - F_{x_{min}}}{F_{x_{max}} - F_{x_{min}}} width$$

$$p_y = \frac{y_{com} - F_{y_{min}}}{F_{y_{max}} - F_{y_{min}}} height$$

The P values are the x and y locations in the pixel grid. The F values are the maximum and minimum values of the range and domain of the area in x-y being mapped to the window, and the Width and Height are the respective width and height of the data window in pixels. For each new point in the trajectory, the corresponding pixel location is generated and stored in the matrix. The matrix is then used to display the trajectory in its entirety throughout the simulation.

In addition to the data window, a textual readout is provided to give more information concerning system parameters. There are static values provided for the user such as the number of shells to be launched, the length of the launch window, and the total mass of the fireworks, but dynamically updating values are also provided. These values are the size of the active mass, the number of shells in the air, the number of burning fragments, and the maximum and minimum height of burning fragments.

5. *OPENGL* FRAMEWORK

All of the graphics utilized in this project was written in C/C++ using the Open Graphics Libraries (*OpenGL*). The Open Graphics Library allows us to model our simulaiton in virtual space by creating a scene into which the simulation is drawn. A proper rendering process is necessary in order to effectively use *OpenGL* and all of its features.

5.1 Graphics Pipeline

Once all of the data for any given scene has been calculated and prepared, it is sent to the graphics pipeline for processing. The graphics pipeline uses this data along with the *OpenGL* code to render the firework simulation. The pipeline is essential because of the nature of frame manipulation. In graphics processing any modifications to a scene are done via matrix operations, and because not all matrix operations are commutitive, the transformation process needs to happen in the correct order [Shirley and Marschner 2009] . Our program also needed the ablility to render multiple layers into one window. Along with the simulation of the fireworks display there is a user interface overlay which portrays information to the user in two dimensions as opposed to the simulation's three dimensions. This is also handled by the graphics pipeline.

Our graphics pipeline is split into four major sections: the main function, the display function, the idle function, and the interface functions. The main function is where the program initializes. In the main *OpenGL* is called in, the window is created, and the display, idle, and interface functions are linked in. The main function, after initializing *OpenGL* and the interface functions, calls the display function. The display function renders the scene using the information provided by the physics calculations. The display function calls a set of subroutines to render the scene, first drawing the fireworks scene, then the fireworks themselves, followed by overlaying two dimensional user interface elements including an on-screen data plot and control buttons.

Our pipeline uses double buffering in order to improve the performance of the rendering process. Double buffering allows for two frames to be calculated at any given time. Once a frame is completed it is pushed to the second buffer where it will be cached and written to the screen. Since the frame is already rendered using a second buffer, our program can begin rendering the next frame in the available buffer while the current frame is being cached and drawn. This aides in not only program performance, but also in the ability to animate seamlessly. In this way we are able to render frames without as much delay for calculations increasing framerates and delivering the user a more smooth simulation.

5.2 Texturing

For this project, we used texturing to our advantage by creating button "textures" and importing them in as a way to create otherwise difficult shapes in two dimensions for the button controls. Texturing in *OpenGL* requires that images used as textures be imported, converted, and loaded into memory. We chose to use RAW image files and used a loading function to process the images and load them into memory as part of the program initialization. The `glTexImage2D()` function simplifies the process with RAW images. This allowed us to set the texture tags as global variables and use them wherever in the code we desired, and also to only load the textures once as opposed to every time the function for rendering the user interface was called. Once the textures are loaded into memory applying the texture to an object is as simple as binding the texture to a surface with the `glBindTexture()` and `glTexCoord` functions.

5.3 Optimization and Tuning

Propagating the fireworks' trajectories between frames puts a significant strain on the processor, especially with a large volume of shells being launched. As such, performance was a primary concern when developing our code. One of the main ways we were able to increase the performance of our program was to utilize the full-screen capability of *OpenGL*. There are three window modes available to us: single windows, sub-windows, and fullscreen [Kilgard]. Single windows mean that multiple windows can be created within one screen. This was disadvantageous for our purposes because for every window a new frame is rendered. This lead to very slow propagation due to the program constantly rendering between windows. Sub-windows push separate windows into one large window, which consolidated windows but still rendered each section of the frame independently. Fullscreen mode, or game mode, captures the entire screen of the computer. Capturing the entire screen means that only one frame is being calculated at every step which increases performance.

When large amounts of fireworks were being launched frame rates still suffered and as a result the propagation of the fireworks appeared to slow down. To counteract the decreased framerates we use an adjustable time step for propagation. Looking at the time between each frame allows us to modify the step value so that instead of slowing down the propagation speed on the screen we increase the amount of virtual time passed between each frame. Self adjusting step sizes allow the program to be used on any computer regardless of hardware and still provide a physically accurate propagation in real time.

6. CAMERA SETUP

In order to provide the ability to maneuver around a scene and manipulate the viewport, a camera must be utilized to represent the viewer within the scene. By default, the camera in *OpenGL* is located at the eye space coordinates *(0, 0, 0)* and looks down the negative Z-axis. By utilizing the `gluLookAt` function from the *GLUT* library, the camera can be manipulated. The `gluLookAt` command consists of 9 parameters of type `GLdouble` [Khronos Group]:

```
gluLookAt(eye.x, eye.y, eye.z, center.x, center.y, center.z, up.x, up.y, up.z)
```

The first three values define the location of the camera in three dimensions, the next three values define a point to look at and the final three define which vector to set as the up vector. In the fireworks simulation, this code is placed in the `reshape` function and utilizes the global variable `eye` of type `Vector3d`. Once the camera is set, the view type is set using the `gluPerspective` function. This function takes four parameters of type `GLdouble`:

```
gluPerspective(fovy, aspect, zNear, zFar)
```

The variable `fovy` specifies the viewing angle in the y-direction and the `aspect` specifies the viewing in the x-direction by taking the ratio of the height and width of the viewport. The final two parameters specify the planes that bound the rendering of the viewport along the z-direction. In order to set the parameters for

`gluPerspective` appropriately for the scene, multiple test cases were performed to ensure that the full scene was encompassed in the viewport.

7. MOUSE AND KEYBOARD INTERACTION

In order to allow for maneuvering around the scene, functionality was added to the keyboard and mouse so that the options to pan, rotate and zoom around the scene. First, the mouse functions were set up so that the user can use the mouse to pan around the scene manually. This is done by utilizing the following functions:

```
glutMouseFunc(void( *callback )( int button, int state, int x, int y))
```

```
glutMotionFunc(void( *callback )( int x, int y ))
```

The `glutMouseFunc` is specified in the `main` method and provides the ability to track the state and motion of the mouse and its buttons. In this simulation, global variables are updated by this function so that other changes can take place depending on their status. Once the variables are updated, the `glutMotionFunc` is utilized to calculate the variation in position of the mouse only when the state of the left button is down. By utilizing these functions together, we have the ability to have a passive mouse in the viewport when not necessary. Next, interaction to move the camera via the keyboard was added to the program. This was accomplished by utilizing function 5 with a customized keyboard function.

```
glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))
```

Similar to the `glutMouseFunc`, this function takes a pointer to a function with the parameters specified in `glutKeyboardFunc`. The keyboard function in is initialized in the `main` and allows for continuous listening for the keys that are pressed by the user. The keys and their respective functions are as follows:

b: Toggle bounding box

c: Toggle center of mass

esc: Quit the program

l: Launch the fireworks

p: Pause the simulation

r/R: Auto-rotate the scene counter clockwise

f/F: Auto-rotate the scene clockwise

q/Q: Zoom out

e/E: Zoom in

s/S: Move up

w/W: Move down

d/D: Move right

a/A: Move left

Once all of these keyboard and mouse interactions have been set up and initialized, the program continues to look for an input. Once an input is received, the variables are modified for their respective functions and are passed to the transformation pipeline. The transformation pipeline is set up in the function called *camera* and contains the necessary matrix functions to modify the scene in accordance to the user's interactions. Prior to modification, the camera is translated to the center of the scene, in this case the launch pad, transformed and then translated back. This must be done to ensure that the transformations are occurring around the correct center point for the scene. The `camera` function is then called from the beginning of the `display` function prior to rendering the remainder of the scene.

8. CONCLUSION

Overall, the combinations of these features creates a realistic simulation of a fireworks display. A mathematical model, combined the graphics capabilities of *OpenGL* are used to create a scene. *GLUT* libraries are then utilized along with self-adjusting code to create an animated scene, with user control and redering in real time. Because the program is modeling a physical system, properties of this system can and are retrieved and displayed to to the user.

REFERENCES

Burden, R. L. and Faires, J. D. 2005. *Numerical Analysis*, 8 ed. Thomson Brooks/Cole.

Fishbane, P. M., Gasiorowicz, S. G., and Thornton, S. T. 2005. *Physics for Scientists and Engineers with Modern Physics*, 3 ed. Pearson Prentice Hall.

Halliday, D., Resnick, R., and Walker, J. 2005. *Fundamentals of Physics*, 7 ed. Wiley and Sons Inc.

Khronos Group. Using viewing and camera transforms. `http://www.opengl.org/resources/faq/technical/viewing.htm`.

Kilgard, M. J. The opengl utility toolkit (glut) programming interface api version 3. `http://www.opengl.org/resources/libraries/glut/spec3/spec3.html`.

Shirley, P. and Marschner, S. 2009. *Fundamentals of Computer Graphics*, 3 ed. A K Peters.

Smith, R. T. and Minton, R. B. 2006. *Calculus: Concepts and Connections*. McGraw-Hill.