Recreating the Amiga Bouncing Ball Demo using C++ and OpenGL

Garrett Grey

May 5, 2021

Abstract

The Amiga Bouncing Ball Demo was a legendary program made to show off the capabilities of the Amiga line of computers. It showed their processing power by having a ball with a rotating checkerboard pattern bounce seamlessly across the screen, and it showed their multitasking power by demonstrating that other programs can run alongside the demo. This power and capability was extraordinary for the mid 1980s. The goal of this project is to recreate this program using C++ and OpenGL. The recreation process involved observing the original program and breaking it down into its basic components to be recreated one by one. Recreating this program in OpenGL was a challenge, and the most import findings from this project was learning how to overcome the difficulties of an unfamiliar architecture, as well as realizing the sacrifices and challenges of front-end programming under a deadline. This project was a valuable learning experience and was helpful to better prepare for a professional position in software engineering.

Introduction

The Commodore Amiga series of computers were powerful machines for their time. The forefront example of this power is their famous Bouncing Ball demo, demonstrating the power of the processor by seamlessly bouncing a ball across the screen with no visible stuttering or slowdown. Even more amazingly, this demo also demonstrated the Amiga’s multitasking power by running other programs along with the demo, an unheard of achievement in home computing at the time. According to an article in the IEEE Spectrum magazine, when the demo was revealed to the public at the 1984 Consumer Electronics Show, the attendees could not believe that such a small computer made for the home market could produce such complex and smooth animations, and some of them tried to find the “real” computer that was running the program [1]. Recreating this program with such a simple description, complicated inner workings, and legendary status would be a perfect sendoff for a four-year educational journey in computer science.

Source Analysis

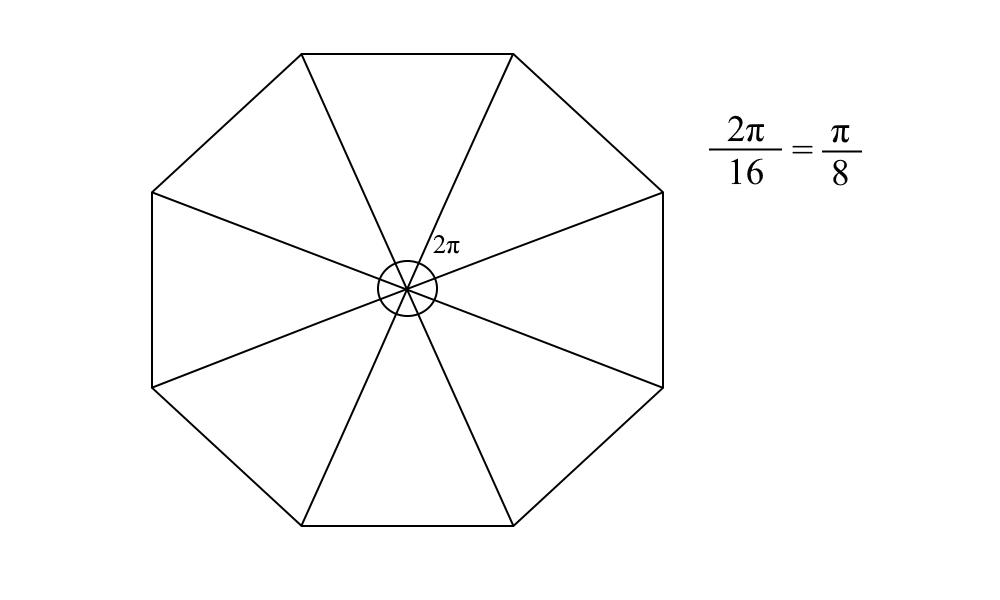
The first thing to do when recreating anything is to analyze it. A recording of the demo running on an Amiga was discovered [[1]](#footnote-2) and the researcher found the following properties:

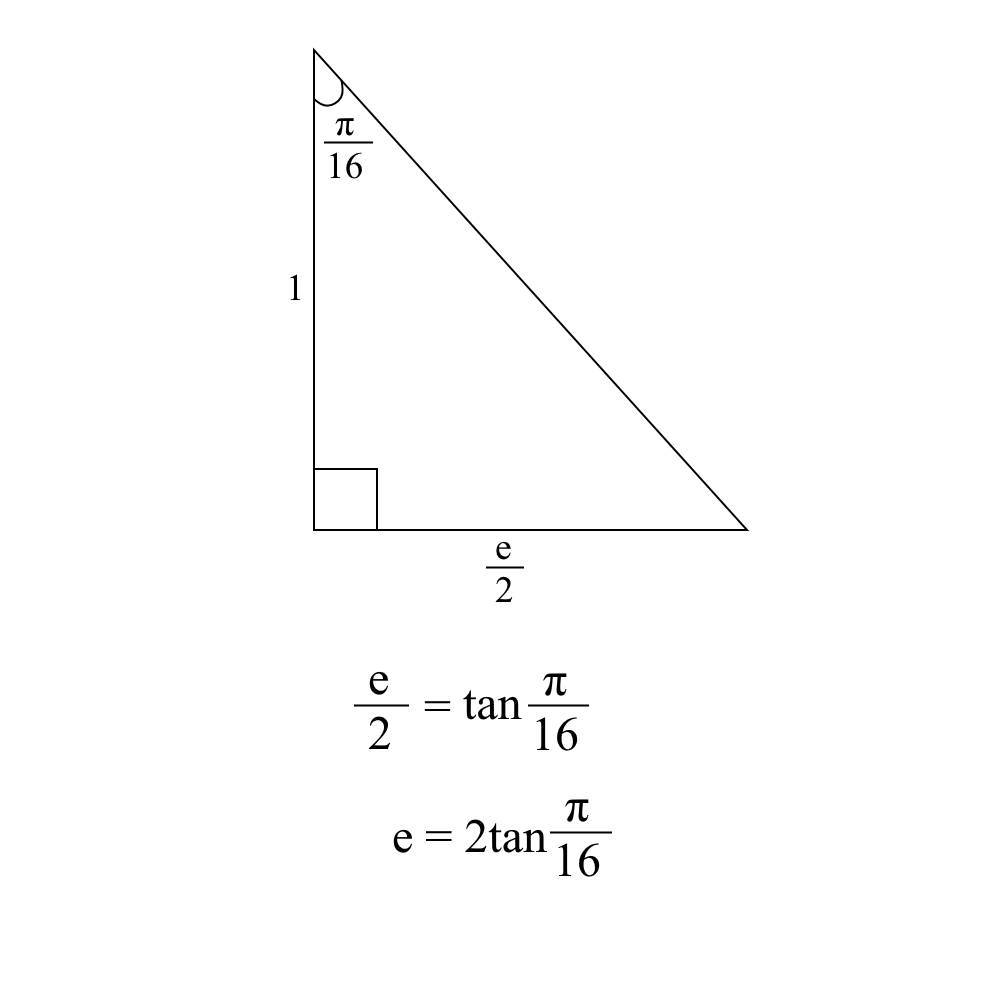
* The ball is bouncing within a purple grid in front of a gray background. The ball is also casting a shadow to its right.
* The grid consists of a floor and a wall.
  + The wall consists of 15 squares in length, and 12 squares in height.
  + The floor is at an angle with the edges on either side seemingly at a 45 degree angle. The angle of the vertices between them seem to converge to 180 degrees in the middle.
* The ball appears to be a 16 sided polygon that is rotated on its right side.
  + The polygon seems to have edges of equal length, and vertexes of equal angles.
  + The ball has a spinning alternating red and white checkerboard texture.
    - The spin is affected by which direction the ball is going, counterclockwise when it is going to the left, and clockwise when it is going to the right.
    - The vertexes of the checkerboard seem to coincide with the vertexes of the polygon.
  + The ball seems to be bouncing at a parabolic trajectory with no loss in momentum, the maximum height and horizontal velocity are constant.
    - A thud sound effect is played whenever the ball either hits the floor.
      * The sound effect pans from the left speaker to the right speaker based on the ball’s horizontal position.
  + The direction of the ball’s horizontal velocity changes when it hits just beyond the side of the grid.
    - The thud sound effect also plays whenever this happens.

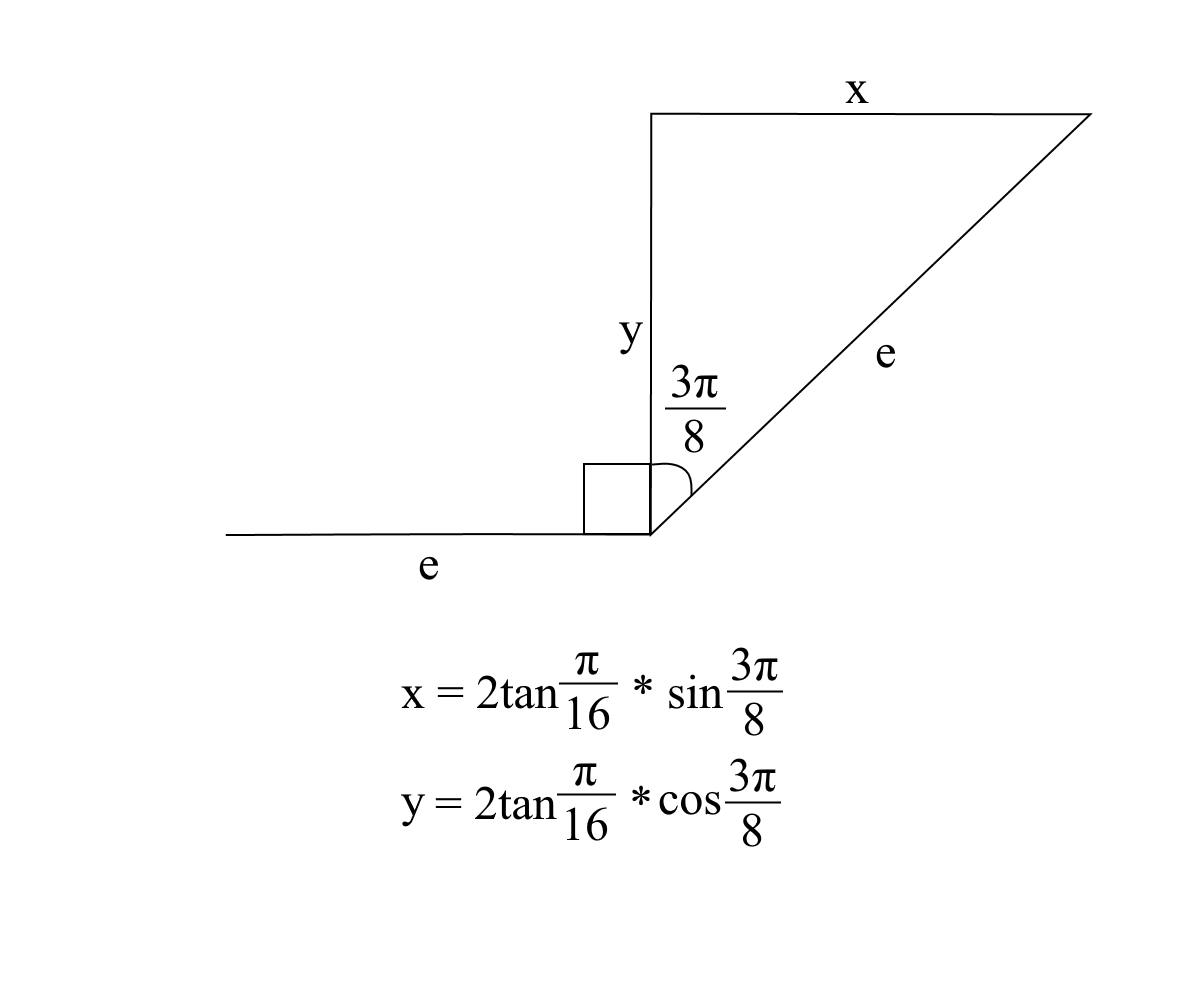
With these analyzed properties, the recreation process can begin.

Recreation

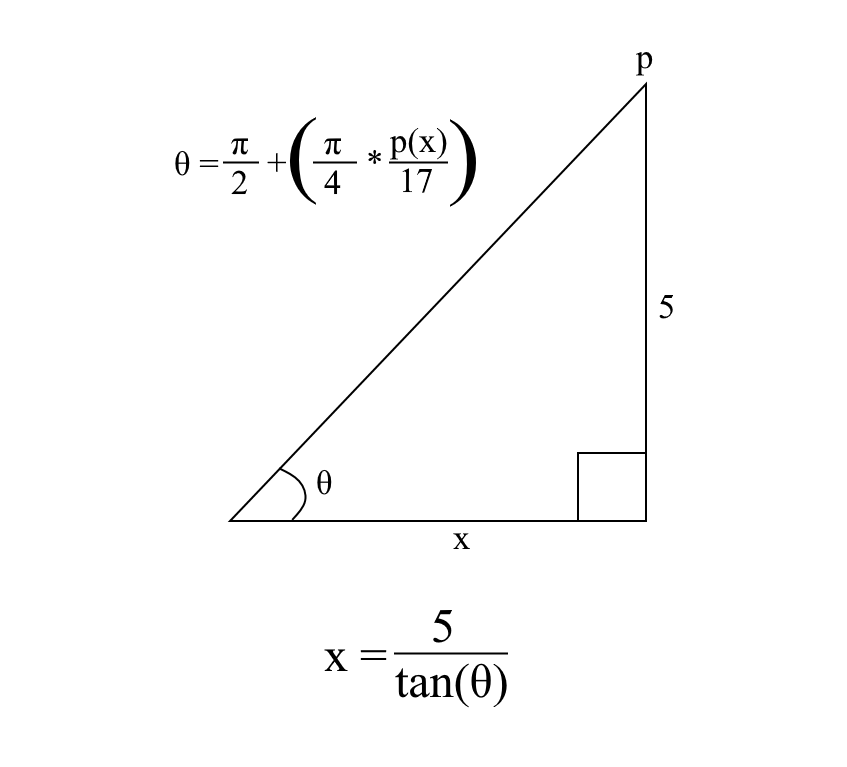
The initial task was to recreate the ball. The first approach was to find out how to draw an equilateral triangle, then a square, then a perfect pentagon, and so on to see how the angles of the vertices changed in relation to their number. This proved to be too time consuming, so the researcher instead changed the approach to finding the properties of an octagon and relate that to a 16-gon. It was noticed that, if a line was drawn from each vertex to the center point, it would result in isosceles triangles, the number of which equal to the number of vertices in the polygon. From this it was concluded that that the sum of all of the top angles of these triangles would be 360 degrees, or 2π radians. (Radians were used from here on because the default input for the trigonometric functions in C++’s cmath library are in radians.) To find the top angle of each triangle, 2π would have to be divided by the number of vertices. Applying this to a 16-sided polygon, the top angle would be π/8 radians. Vertically splitting one of these triangles in half would result in a right triangle, allowing the use of trigonometric functions to find the length of each edge. This also set the top angle of the right triangle to π/16. Finding the bottom edge of this triangle was the goal, because it coincided to a half of the length of the edge of the polygon. The function needed was tangent, because the tangent of the angle is equal to the opposite edge divided by the adjacent edge. The adjacent edge is arbitrary as it represents the radius of the polygon and only affects its size. As such it was set equal to one for the sake of simplicity. This would mean that one half of the length of the polygon’s edge is equal to tan(π/16), so the total length of the edge is 2\*tan(π/16).



Next was to find the diagonal vertices. To find the number of these vertices, and thus the number of edges, 16 had to be subtracted by 4, the total number of horizontal and vertical edges. Dividing this number by 4 results in the number of diagonal edges between horizontal and vertical edges, which is 3. Conveniently, the researcher already set up a way to find the angle of the first diagonal edge after the horizontal. Going back to splitting the 16-gon into isosceles triangles, the two lower angles would represent one half of the needed angle, and would be equal to each other. The top angle was already found, so it was a simple matter of subtracting that by π. (Dividing by two to find the individual angles wouldn’t be necessary as that number would be multiplied by two to find the angle needed.) This resulted in an angle of 7π/8 radians. The next step was to find the change in the x and y coordinates between the two points. This involved subtracting the angle by π/2 and using trigonometric functions on the result along with the length of the edge. In the end, this resulted in a difference of 2tan(π/16)\*cos(3π/8) in the y coordinate and a difference of 2tan(π/16)\*sin(3π/8) in the x coordinate. Applying this to the rest of the vertices allowed finding the rest of them between the horizontal and vertical edges, and polarizing the results allowed finding all the other vertices in the polygon. To translate this into code, a starting point needed to be defined (the right point of the lower horizontal edge) and these changes would be applied individually, later condensing them into for loops for a cleaner presentation.



Next, the the grid background was recreated. The wall of the grid was simple, all that was needed was to define the points on the y axis where the grid would begin and end (which would be -17 and 17 where the minimum and maximum were -20 and 20), find the distance between the two, divide that by 15, and set those resulting points as places to draw vertical lines. Repeating the process on the horizontal axis resulted in a 15 by 12 grid. The difficult part was the floor. The left and rightmost edges of the grid were estimated to be at a 45 degree (π/4 radian) angle, but the angles of the lines between the edges approached 180 degrees (π radians) as they reached the center of the grid. The approach the researcher used to find these angles was by taking the x coordinate of the start point *p*, dividing that by 17, and then multiplying that by π/4 to get the angle. This would result in the angle being π/4 at the edges, π in the middle, and naturally converge between them. From there the change in the x and y coordinates of each point would be found using trigonometry. The change in y coordinate would have to be constant, as each line ends at the same y coordinate. Manually adjusting this led to 5 being the best value. From here, the change in x would be found by dividing y by the tangent of the angle.



After completing that task, the ball’s bouncing algorithm was recreated. The ball would be given an upward velocity once it had passed a certain point representing the floor, and the velocity would be subtracted by a constant amount if it was above that point, eventually becoming negative and returning to the floor. This would simulate gravitational force. Every time the function would be called, the velocity value would need to be added to the ball’s middle point. This would make the ball bounce vertically. Making it bounce horizontally was much more simple. Because there were no forces acting on it on that axis, its velocity would remain constant. The only change it would go through would be that its polarity would alternate any time it was on a point beyond the grid.

Texture mapping in OpenGL is a complicated process. The researcher had no experience working with OpenGL before this year. A Computer Graphics course, which was completed in conjunction with this project and taught OpenGL, did not cover texture mapping, so the researcher had to explore the subject alone. The basics of setting up textures is easily searchable on the internet, but a way to fit it onto a predefined object could not be found. This, on top of time constraints, led the researcher to conclude that this aspect could not be implemented into the project. Perhaps a texture could have been simulated by having the ball be constructed by a series of trapezoids, and changing their color on each point at a precise area to simulate a checkerboard, and changing that point to simulate the checkerboard’s movement. That, however, would require a complete overhaul of the construction of the ball, and with the time restraints it was decided to carry on with the ball being a solid red. This, however, is one of the many difficult decisions one needs to make in front-end programming. Deadlines are set whether the product is ready for release or not, and working on an aspect that is not critical to the project’s main goal is wasting time on working on aspects that are.

From here, certain aspects of the program were tweaked, such as the ball’s velocity and its size, so that it looks and functions more closely to the original demo. It was here where the ball’s shadow was added. In the original, it looked as though the shadow was just a dark and transparent copy of the ball just behind and to the right of the actual ball. To accomplish this, the code that rendered the right half of the ball was copied and put it in a new function, and then called that function after the grid function and before the ball function to have the program render the objects in that order. To make the shadow partially transparent, blending had to be enabled and the blending function had to be set to use the alpha channel. This would allow the alpha channel to act as a percentage of transparency when declaring the shadow’s color. After this the researcher attempted to rotate the ball on its side, but this conflicted with the bouncing function, rotating the ball’s velocity as well. Where it would normally bounce vertically, it would bounce diagonally towards the left side of the screen. This could have been remedied by either redrawing the ball so that it appeared on its side, or by modifying the bouncing function by translating the ball’s velocity using trigonometry. However, both of these solutions would be complicated and time consuming, and with the time constraints, this aspect had to be omitted as well. Adding the iconic sound effect from the original was debated, having it pan from the left speaker to the right speaker depending on the ball’s horizontal location, but this likely wouldn’t have been apparent during the virtual presentation and was thus omitted.

Results

The final program is similar to the original Amiga Bouncing Ball demo. It features a purple grid cast on a gray background, with a solid red ball bouncing within and casting a shadow on the grid. There are two main features missing from the program, being the rotating checkerboard texture on the ball, and the fact that the ball is not tilted to its right side. These two features were omitted due to time constraints and technical problems. If work were to continue on this project, these two aspects would be the first things to focus on. Adding sound would be the final addition in terms of the recreation itself, possibly followed by detecting the screen resolution of the host computer and adjusting the program window accordingly along with other ease of use tweaks.

Conclusion

Recreating the Amiga Bouncing Ball demo was a valuable learning experience. It was helpful in understanding the nuances and difficulties in programming with low-level computer graphics, and it brought to light the difficult process and sacrifices involved in front-end programming. Many planned features had to be omitted from the final product in order to have it released on time, meaning the most important features must be prioritized in order to create a more ideal and functional product. This is a tough lesson to learn, and is very beneficial to learn it early on in a programming career.

References

Cited in IEEE format.

[1] P. Wallich and T. Perry, "Amiga: The Computer That Wouldn’t Die", *IEEE Spectrum: Technology, Engineering, and Science News*, 2001. [Online]. Available: https://spectrum.ieee.org/computing/hardware/amiga-the-computer-that-wouldnt-die. [Accessed: 22- Apr- 2021].

1. <https://youtu.be/-ga41edXw3A?t=27> [↑](#footnote-ref-2)