The Creation of a Bullet Hell Game Engine

Curtis Mackie
Information Science, Technology and Mathematics Departments
Doane College
Crete, NE 68333
curtis.mackie@doane.edu

Abstract

Bullet hell is a type of shooting game characterized by large quantities of bullets. This presents a computational challenge for developers, who must find a way to process hundreds of bullets in a very short length of time. For my senior research project, I designed and coded a game engine for handling the unique requirements of bullet hell games. I used the RADIS approach to efficiently design the engine. The work-in-progress engine is coded in C, using SDL for graphics and Lua for scripting. Although much remains to be done, many features of the engine can be demonstrated, including its collision detection and rudimentary scripting capabilities.

1 Introduction

Bullet hell, as a genre, is not particularly well-defined. In some ways, it is an offshoot of old-fashioned “shoot 'em up” games like *Gradius* and *Galaga*, but with a completely different focus (Carter). As such, bullet hell games, though superficially similar to old-fashioned “shmups,” *feel* very different to play. This is due to some major gameplay deviations bullet hell games have made from their classical roots over the years (Carter). Most notably, bullet hell games derive their name from the large waves of bullets they present to players (Carter). This, in turn, motivates other gameplay features, such as the small hitboxes many games are seen with.

Bullet hell got its start with the arcade game *Batsugun* (Carter). While *Batsugun* wasn’t particularly similar to today’s bullet hell games, it began pushing the envelope of what arcade hardware could do, and how ridiculous a shooting game could be (Carter). Later, companies like Cave and Treasure would codify the budding “manic shooter” genre, more commonly known as bullet hell today (Carter).

Before beginning the meat of this paper, a few important terms must be defined. First, *collision detection* is the process by which a game determines which pairs of objects are touching and which are not (Ericson, 1). Because it is impractical to determine this with 100% accuracy, they usually employ a *hitbox* (or, more formally, a *bounding volume*), which is an abstract construct that roughly approximates the shape of the object (Ericson, 75). It uses this hitbox to decide what space in the game world that object occupies. If two objects' hitboxes are intersecting, then the two objects are attempting to occupy the same space, and therefore collide (Ericson, 75). It is important to note that hitboxes need not be an actual box: boxes, circles, polygons, and even more complicated constructs like half-space intersections all see use as “hitboxes” in video games, with some shapes being very efficient for certain applications, and almost useless in others (Ericson, 75-103).

Next, a *game engine* is, roughly speaking, all of the code that is responsible for running a game, minus all of the details that make a game unique (Gregory, 11). This way, you can change some of those details to make a new game, without changing the engine at all (Gregory, 11). An ideal engine allows users to make any game they want without modifying the engine (Gregory, 11). For example, you could easily make a bullet hell game in the Unreal 3 engine, even though that engine was ostensibly designed for first-person shooters.

I employed the RADIS framework in the development of my project. RADIS is a generic problem-solving template that stands for Research, Analyze, Design, Implement, Support. These are steps commonly used in the information technology field when creating solutions for clients, so I felt it appropriate to use them here. The remainder of this paper details the development of the project through all of those steps.

2 Research

When researching this project, I first looked into already existing bullet hell engines, to get a feel for what features my project might be expected to have, as well as what features I would like to implement that other engines ignore. I was already aware of Danmakufu at the start of this research, and I learned about another one, BulletML, along the way.

Danmakufu is a freeware program that allows users to script their own bullet hell stages ("Touhou Danmakufu"). The scripting language is very powerful, supporting, among other things, 3D backgrounds, particle effects, and unlockable content ("Touhou Danmakufu"). It also makes many of these features work with just one line of code, whereas a programmer working from scratch would require thousands of lines of code to replicate the effects. Entire professional-quality games can be developed using this program, and have been in the past ("Concealed the Conclusion").

However, Danmakufu is not without its faults. One of the most notable faults is the interface. Even if you build a complete game, with its own menu, you still have to go through the standard Danmakufu interface to launch it. While the interface was intended to make it very easy to debug your own scripts, it is also confusing for the end-user to launch someone else's, due to vaguely defined terminology for different types of scripts. The scripting language itself, while perfectly usable, has some strange function conventions, such as numbered functions like "Explosion01" with only one version. Most importantly, the source code is not available, which means it is unusable as an engine; if a desired feature is not available in the engine, and it is not easy to work around it, the only thing to do is hang your head and come up with something else.

BulletML is an XML-based engine created by Kenta Cho (Cho). The engine uses XML files for scripts (Cho). This has the advantage of allowing bullets to be nested into groups, which is intuitive for many bullet patterns. Furthermore, the engine is fairly fast, and while it lacks many of the features of Danmakufu, some of these can be added to the open source code.

However, this turns out to be its main downfall. Limitations brought on by the XML-based language (in particular, the inability for bullets to react to the player) are crippling, severely hampering one's ability to design for this engine. While certain kinds of games will not notice these limitations, more involved bullet patterns will run into these walls hard. As stated before, some of these can be added to the engine directly, but the engine itself is programmed in D, a relatively obscure language which is not used in many serious applications. As such, most programmers who would like to work with this engine will probably not have prior experience with D. The engine also requires certain things to be hard-coded into the game, so actually making a game using the engine will require some tinkering with the code either way.

The biggest concern with BulletML, however, is the collision detection. The engine always uses axis-aligned bounding boxes, making no exceptions. Axis-aligned bounding boxes (AABBs) have the advantage of being very fast, with only a few comparisons needed to test for collision, but they are also extremely inaccurate for bullet hell games.

Most bullets are round, or at least best approximated by circles. For very large bullets, an AABB would massively under- or overestimate the area of a bullet, resulting in wide areas where the player should collide, but fails to do so (or vice-versa). This is not a theoretical concern; Utsuho Reiuji, the final boss of *Touhou Chireiden*, uses bullets that are nearly the size of the screen in many of her attacks. (Collision detection for circles and axis-aligned bounding boxes will be discussed in more detail in the Implementation portion of this paper.)

The other portion of my research was looking up algorithms and designs I would need to build a game engine. I primarily read through two books, *Real-Time Collision Detection* by Christer Ericson and *Game Engine Architecture* by Jason Gregory. Based on this research, I was able to come up with a reasonable set of features that my project should support, as well as understand how to implement those features.

3 Analysis

For the analysis step, I mainly concerned myself with coming up with a set of features that I wanted. The most important requirement I decided on was making my engine fully data-driven. This means that an entire game can be created using the engine without having to touch the engine code itself. For this to happen, the structural design of the game (such as level layouts and resource filenames) must be entirely specified in external data files, and not hard-coded into the engine. It also means that the engine must provide a scripting mechanism that can be used to program game logic which is independent from the engine, in particular the behavior of enemies and bosses.

This one decision dictated many of my other requirements. Lots of resources now have to be stored apart from the game code, requiring a resource management system that loads resources from archives. Scripts have to have a way of communicating with the engine, requiring some kind of unified scripting system, and greatly influencing how enemies and bullets are stored as they have to remain associated with their scripts.

From here, I considered what kind of focus I wanted my project to have. While it might seem that all bullet hell games are created equal, there are many distinctive differences in how these games present themselves. For example, games in the *Touhou* series tend toward a more aesthetic focus, with vibrant color schemes, diverse and low-tempo soundtracks, and intricate bullet patterns with complicated choreography, while games like *DoDonPachi* and *Mushihimesama* are much more gameplay-oriented, with distinct monochromatic bullets, fast-paced electronic soundtracks, and dense patches of bullets with simple movement patterns. These two games appeal to two different crowds of gamers, and have very different scripting and resource management requirements.

In the end, I decided on the latter approach, for a number of reasons. First and foremost, the general system requirements would be lower. Bullet patterns in the *Touhou* series incorporate bullets that curve, loop around, speed up or slow down, and generally behave in unusual ways, and while this is visually interesting for the player, it means that every

single bullet has to have a script telling it to do this. Scripting bullets is not computationally cheap; a scripted bullet must execute arbitrary script commands, while a normal bullet need only update its position by its velocity, which is a simple addition instruction.

Furthermore, the more intricate bullet patterns also require a more involved scripting engine, and I did not think I would have the time to implement this. For example, it may seem easy to make a bullet constantly change its direction to "seek out" the player, but this is actually an involved process: the program has to find the direction of the vector between the bullet and the player, and replace the bullet's velocity vector with some vector between its current angle and the calculated direction. Not only is this computationally expensive, it is not trivial to program. Deciding not to implement these more complicated operations simplified this part of the design greatly.

4 Design

The design phase of this project was plagued with over-engineering. There were many problems that needed to be solved, and quite a few of them received far grander and obtuse solutions than they required.

One significant example was in designing the collision detection system. One initial design (indeed, the entire point of the original project pitch) was to use bounding volume hierarchies. This is a collision algorithm that involves constructing a binary tree of hitboxes (Ericson, 235). Each node in the tree is a simplified hitbox that surrounds its two children entirely, all the way down to the leaves of the tree, which are the actual objects we want to test (Ericson, 235). Because of this construction, when testing for collision on a bounding volume hierarchy, we can eliminate entire subtrees if their root member does not collide with the object we're testing against, thus saving a substantial amount of time (Ericson, 235).

Bounding volume hierarchies are typically used in applications where multiple objects will need to be compared against each other – one bounding volume hierarchy is checked against another. In these scenarios, the number of potential collisions goes up as the square of the number of objects (in computer science terms, the algorithm is in $O(n^2)$), so the hierarchies will save an enormous amount of work.

This is simply not the case in our bullet hell engine. We might have many thousands of bullets on screen, but there will still be only one player. Thus, instead of $O(n^2)$, we have a linear, $O(n)$ relationship. Building and working with the hierarchy would be much slower – somewhere around $O(n \log n)$, depending on how long our heuristics take to calculate – and would not even be guaranteed to reduce the number of collision checks we would have to make. The naïve approach of simply checking every bullet one at a time will be much faster most of the time.

Another common setback that I encountered was predicting what features might be

required. As an example, early on, the resource system would have allowed individual resources to be loaded from an archive, rather than loading the entire archive at once. This turned out to be completely pointless, as needing to load individual files from an archive usually just means the archives were organized poorly. With proper archive organization, it is possible to eliminate both unnecessary resource loading and archive overlap, while also avoiding the long lists of resources that loading files individually would require.

Not helping this is the high interdependency between different systems. Each system relies on others in subtle ways, and it may not be clear what features another system will require until you've finished. For example, when I was doing some work on the "example" ship that would be used in demonstrations, I found that I needed to keep track of some variables that could be attached to the object, which resulted in adding an array of registers to each player object.

The final design contains anywhere from nine to twelve systems, depending on how one defines a "system." The bullet, fixed-point arithmetic, initialization, input, menu, player, resource, scripting, and timer systems all have clearly-defined and separate purposes. However, there are also source files for collision, debugging, and geometry, which some might classify as "systems," although they are mainly just there for organizational purposes.

5 Implementation

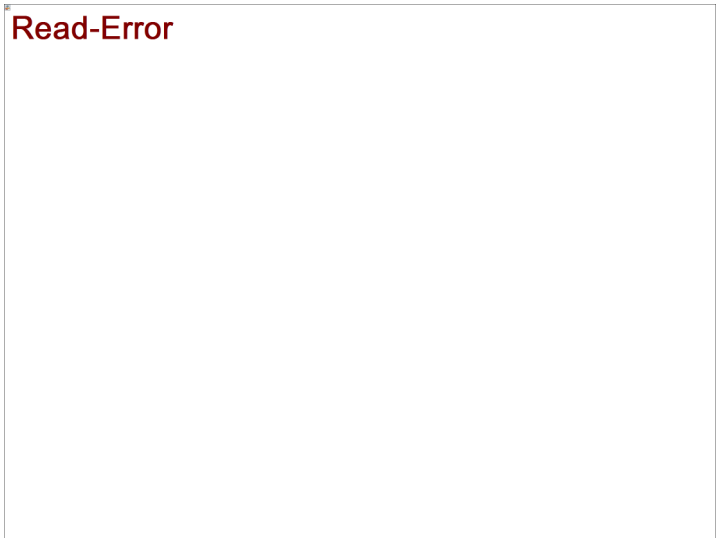

Figure 1: Engine test.

I could very easily make this section as long as the entire paper, going into excruciating detail of how every single system is implemented, and the different phases it went

through as it evolved into the finished product. This, however, would be completely uninteresting. Instead, I will focus on the mathematically-oriented facets of the engine. In particular, I would like to take a look at the fixed point number system and the collision detection system.

5.1 Fixed-point numbers

In computer science, there are two ways of working with numbers that have a non-integer component. The most common method is with floating-point numbers, or just "floats" for short. At their core, floats consist of multiple components, which are combined in a manner similar to scientific notation (Goldberg). Since the exponent varies the scale of the value being represented, the decimal point in the final value can move around, or "float," hence the term "floating-point number." Floating-point numbers are extremely common, with almost every programming language providing support for them (Goldberg).

However, floating-point is not the only way to go. Instead of allowing the decimal point to float around via an exponent value, it is possible to arbitrarily place the point at a single, known location for all numbers. Doing so results in the *fixed*-point number. This construction greatly constrains the range of possible values that can be represented, but it has several advantages. Most importantly, because fixed-point numbers are essentially integers multiplied by a common factor, integer operations can be used to work with them, and computers are much faster working with integers than with floating-point numbers. As such, fixed-point numbers are a superior choice for this project.

Fixed-point numbers are used for all geometric representations. The (x, y) coordinates, radii, and velocity vectors of all bullets are stored as fixed point numbers. This may seem unnecessary at first, since pixels, by their very nature, are not divisible. It does not seem sensible to keep track of portions of a pixel, when the pixel is the finest unit the screen is capable of displaying anyway. However, it becomes important when considering objects in motion. Velocity is measured in pixels per frame, and a frame is $1/60^{th}$ of a second, so if the engine doesn't have a way of measuring distances smaller than a pixel, the smallest possible speed (besides zero) would be 60 pixels per second. This is clearly absurd. Keeping track of "subpixels" solves this problem, and by using fixed-point arithmetic, very little time is wasted doing so.

Fixed point numbers in this project are represented using a "Q15.16" system – 15 bits before the point (the most significant bit stores the sign and is not counted), and 16 bits after. This means that the possible range of values runs from -32768 and -65535/65536 to 32767 and 65535/65536. This may not seem like a large range, but it is more than sufficient for this engine; there is absolutely no reason one should ever need to keep track of bullets over 30,000 pixels beyond the edge of the screen.

5.2 Collision detection

With this, we now have all the geometric measurements we need to run collision checks. This engine uses only two collision calculations, circle-to-circle and AABB-to-AABB. These are two of the easiest (and fastest) calculations to use, and are therefore ideal for this kind of project. Both calculations are similar in that they take two shapes and determine if they are intersecting or not; the only difference is what kinds of shapes are being checked.

For a circle-to-circle collision check, we need two circles, obviously. Each circle is described by three quantities, the *x* and *y* coordinate of its center and its radius. Actually determining whether the two circles are intersecting is fairly simple; we just need to calculate the distance between the centers, and compare it to the sum of the radii. If the distance is greater than the sum of the radii, the circles are not intersecting; otherwise, they are. Using the distance formula, we get the precise comparison, $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2} \leq r_1+r_2$. However, because square roots are very slow to compute, it is generally advisable to avoid them. In this case, since both quantities will obviously be positive, we can square both sides of the inequality without affecting its direction, giving us the revised formula, $(x_1-x_2)^2+(y_1-y_2)^2 \leq (r_1+r_2)^2$. The player and all enemy bullets have a circular hitbox, so this circle-to-circle calculation is used for collisions between the player and enemy bullets.

The algorithm for two AABBs (axis-aligned bounding box) is much simpler. An axis-aligned bounding box is a rectangle whose edges are parallel to the axes. More usefully, an axis-aligned bounding box can be thought of as the set of all points (*x*, *y*) with $x \in [a,b]$ and $y \in [c,d]$ for a given *a*, *b*, *c*, *d* (Ericson, 79). As a result, if one AABB is given by the intervals $[a_1,b_1]$ and $[c_1,d_1]$, and the other by $[a_2,b_2]$ and $[c_2,d_2]$, then the boxes will intersect *iff* $[a_1,b_1]$ and $[a_2,b_2]$ intersect, and $[c_1,d_1]$ and $[c_2,d_2]$ intersect (Ericson, 79). This can be determined by the logical statement $b_1 < a_2$ or $a_1 > b_2$ or $d_1 < c_2$ or $c_1 > d_2$, which is false *iff* the boxes intersect.

Axis-aligned bounding boxes are used for collisions between the *player's* bullets and enemies. This is because, while there aren't necessarily a lot of player bullets on screen at any one time, each will have to be individually checked against every enemy on screen. Since this means the number of necessary checks can increase very quickly, the circle-to-circle algorithm suddenly seems slothlike with all its multiplication. Since checking AABBs requires only eight comparisons (and that only in the worst case!), it is clearly a much better choice here.

6 Support

While I have completed a large portion of the engine's essentials, the engine is still far from finished. There is an enormous amount of work that still needs to be done, which I

will briefly describe here.

Most notably, the Lua scripting system is barely started. It is not actually possible to create a game using this “engine,” as the system test is the only part that's coded. Being able to load the resources for a specific game and play it is fairly important for a game engine, and the scripting engine will have to be finished in order to do this. I also plan to implement multiple “polish” features, such as music and sound effect playback, particle effects, and support for certain common features like high scores tables.

Finally, there's the issue of documentation. I've begun documenting parts of the engine's code on the project's Google Code wiki, but there's far, far more to be done. The code interfaces for the bullet and player systems haven't been documented at all, and even early work like the geometry functions is still unexplained. Documentation is very important, especially for an open source project, so this needs to be dealt with at some point.

References

Carter, Chris. “Gamer’s Dictionary: Bullet Hell.” Gamer Limit. 17 Feb. 2011. 29 Nov. 2011. <http://gamerlimit.com/2009/02/gamers-dictionary-bullet-hell/>.

Cho, Kenta. “BulletML.” ABA Games. n.d. 30 Nov. 2011. <http://www.asahi-net.or.jp/~cs8k-cyu/bulletml/index_e.html>.

“Concealed the Conclusion.” Touhou Wiki. 3 Nov. 2011. 30 Nov. 2011. <http://en.touhouwiki.net/wiki/Concealed_the_Conclusion>.

Ericson, Christer. Real-Time Collision Detection. San Francisco: Morgan Kaufmann, 2005.

Goldberg, David. “What Every Computer Scientist Should Know About Floating-Point Arithmetic.” Oracle. Mar. 1991. Sun Microsystems, Inc. 29 Nov. 2011. <http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html>.

Gregory, Jason. Game Engine Architecture. Natick, MA: A. K. Peters, Ltd., 2009.

“Touhou Danmakufu.” Touhou Wiki. 3 Oct. 2011. 30 Nov. 2011. <http://en.touhouwiki.net/wiki/Touhou_Danmakufu>.