First Person Strudel: Toaster Genesis

CS335 Technical Report

Lizandro Perez

Charles Enright

Roy Banuelos

Nicholas Gardner
Intro

Our game is called First Person Strudel: Toaster Genesis, and it is a 3D first Person shooter game with many amazing features. The game required sounds, physics, math and a lot of hard work. Each team member had responsibility of contributing a functional section of the game and collaborating to create a masterpiece of a game. Lizandro covered sound and 2D features including high score, Nick did collision, walls, and miscellaneous features such as high score and game over. Roy created models, textures, and sky, and attempted to integrate his 3D World Engine into the source. Chad covered enemy AI and miscellaneous features such as game over.

Game Premise

The game places you in the center of a small, non-linear map with several armed velociraptors. These raptors are able to cause damage to the player both by shooting at them and by attacking them. To combat these raptors, the player has three guns available with different power, range and clip sizes. The goal of the game to is kill as many raptors as possible before your health drops to zero. Some key strategies to do well are to keep distance from the raptors so that their inaccuracy is exaggerated and to take cover while reloading.
Sound and 2D effects

Lizandro’s contribution to the game consisted of creating sounds for the game. He created different sounds for each of the weapons option we offered. Lizandro created sounds for a shot gun, sniper, and a 9mm handgun. Sound features were also integrated into the game, such as the sound of firing a gun on an empty clip. Every time the user tries shooting but there is no ammo left in the clip, a reload prompt is triggered and displays to the screen and the sound file for an empty clip is played. The user can reload by pressing spacebar to reload the ammunition into the gun. Lizandro also was in charge of taking care of the weapons settings. Based on the weapon selected Lizandro set a max amount of bullets and kept count of the bullet count after each time the gun was fired.
Another feature Lizandro made to the weapons was based on the current weapon and the distance between the shooter and the target it would establish the amount of damage the weapon would produce. Lizandro modified the shotgun such that it would be effective only at close range. If the player was shooting a target in the far distance the damage of the shotgun would be minimal; however the sniper rifle’s damage was not changed much by the distance. Sniper rifles still have a lot of power even when shooting from a distance.

Lizandro also added a game menu. He created a menu that keeps tracks of the amount of kills the player has in the current game session. It also displays what keys to press to trigger each team members key feature. Another item it displays is how many bullets it has left out of the max bullets allowed for that gun. Finally, he also has it displaying which key to press to select a certain gun and which keys to use to shoot or to reload.
For Lizandro’s special key he created a sniper scope that is displaying the distance to the target and which target number we currently are trying to kill. In the sniper scope it is also displaying how many bullets are currently left in the chamber with a text image and as each bullet was fired a text bullet was removed. When a target is hit the cross hairs of the scope will turn red. The scope can stay active as a zoom scope for the rest of the weapons as well.

Lizandro also created a leaderboard. When the game ends the number of kills will be compared to the top 5 scores; if the score is larger than any of the current top 5, the person will be added to the leaderboard unless their player name is one of the magic values “test” or “player”. The board will automatically be calculated and rendered at the end of the game session. The scores are kept and stored in a database and when the game is over it will pull the information, store it in an array and then compare the scores. If the score is greater than or equal to a current score, the new score will replace the old score.
A major task Lizandro completed was to create weapon settings. One setting is a reloading function to keep track of current and max ammunition for the current weapon. Another setting is to calculate the amount of damage a weapon would create base on the distance. For the sound portion, Lizandro created sound sources for each of the weapon selections and a sound for an empty clip. At the end of the game session the sound sources were erased to free up the space. The menus Lizandro created were just to display which keys have functionality for our game and also the leaderboard that will be displayed at the end of each game session.

Mapping and walls

The game’s map was designed in the software “Dia’ and exported to plain SVG, which is a vector graphic format. When the program starts, the SVG file is loaded and interpreted into walls. While it appears 3 dimensional, the map is actually stored entirely as 2 dimensional line segments with wall height being the extent of a third dimension. To prevent enemies from spawning outside of bound, spawn points are put in the SVG file as specifically colored line segments, shown below as small purple lines.
Enemies are continuously spawned as old ones are killed. To prevent awkward or confusing gameplay, new enemies are ideally spawned outside of the area of the player. To accomplish this, when a new enemy is spawned, a random spawn point is selected then checked to be within a certain distance of the player, as shown above. If it is within that distance, a new one is chosen until an acceptable point is found. If such a point is not found after 20 tries, any point is chosen to prevent lockup.

Wall collision detection for both the player and enemies is done on two dimensions using some basic vector calculations. First the problem is flattened down onto a single plane so though there was no height, simplifying the wall down to a single line. Next, the closest point on that line to the player or enemy (P) is located using the calculations shown below. Now we can determine whether or not there was a collision by checking if the distance between P and the closest point is less than the radius of the player or enemy. This is done for each enemy and player for each wall. While this sounds computationally expensive, the game is able to handle tens of thousands of these calculations each movement without any lag and only actually requires less than one thousand.

For shooting detection, ray-sphere collision is used to determine if the player or an enemy has been shot and ray-plane intersection is used to determine if there is a wall in the way. When a shot is fired, a ray is created with an origin at the player and in the direction they are facing. Additionally, a ‘closest collision’ float variable is created.
and initialized to a very high value. This ray is then tested on all walls. Each wall performs a ray collision test which involves testing for a ray-plane collision, finding the distance to the collision then checking that the collision is not over, under or to the side of the edges of the wall. If the distance is larger than an existing collision, then the wall is ignored. Otherwise, it updates the new closest collision variable.

Next, the same process is done on all enemies with ray-sphere collision. If a collision is detected, and that collision is closer than the closest wall collision then it is considered a hit. After all enemies are tested and the closest hit is determined, the a damage function is called on that enemy. The value of the damage is based on the power of the current selected gun and the distance of the collision.

Texture fonts were added late in the game development. The texture fonts use a sort of sprite sheet and a monospaced font to simplify the placement of strings of text. Each ‘cell’ on the sheet corresponds to the ASCII table starting at character 32. The texture font process is made up of two functions. The first function takes a character, finds the coordinates of that character on the font sheet, then displays it at the coordinates and scale specified. The second function takes an entire string, some coordinates and a height and determines where each individual character goes, calling the first function for each. This second function can even center or right align the text.
Mob Movement and AI

In order to create an interesting play experience, the game is populated with destructible elements called mobile objects ("mobs"). Our original model for the mob extended Roy’s Sphere class, integrating its isTouching method into mob collision. These spheres, which also represent each mob’s collision surface regardless of how they are rendered, can be toggled on and off by hitting ‘c’ (the Chad key). The Chad key also lowers the height of each wall so that all mobs are visible (though the mobs will still collide with the walls regardless). Finally, while they are spheres the mobs’ weapons are disabled and they will not shoot the player. This is because, although movement collision assumes the walls are of infinite height, bullet collision does not and it is possible to shoot (and be shot at) by every mob when the Chad toggle is set. This also allows the player to shoot mobs that are in an illegal position, such as boxed in by walls or outside the play area.

Now that we have mob spawn points and AI properly set, this is less of a problem, but this functionality was of great assistance when the team was developing mob presence and behavior. For example, it allows the developers to notice when a mob has violated the play area and is wandering off, which can sometimes happen if a mob thinks it is stuck and decides to teleport. Velociraptors only teleport when A* is toggled on (toggled via the ‘m’ key) and no solution is found to the A* path.

The Chad key also toggles the game’s temperature attribute from fahrenheit to celsius and back.

Mob A* is based on CS312 lab 6, the Dijkstra’s algorithm. When a mob is created, it selects a random offset between 0 and 120, and then begins counting. When its count reaches 120, if A* is toggled on, the mob attempts to find a path between its current position and the player’s location.

One restriction the A* algorithm has is that it must operate on a map. Because each mob will be marking its A* map every time A* is called, each mob has its own special map that it is allowed to mark up. This map takes as input a pointer to the game
object, and calls each wall to give it a set of points that represent each “square” that each wall occupies. For purposes of speed optimization, each set of points is reduced by half, producing a map that is 1:2 scale and an array of arrays that is 1/4 as large as the actual game area. The height attribute of the map, mob, and player are discarded. Walls are also offset; in the game, the walls are centered around 0, 0 with some having negative values and some having positive values, but for the A* map each wall occupies a positive index of an array, so we simply add 50 to each coordinate to offset. Each wall is marked on the map as an impassable obstacle. The map is created only once for each mob, generally when the startAstar function is called, and it assumes that the position of the walls will never change. One of the early design decisions the team made is that although we have a destructible wall class, we aren't using it and therefore walls are never created or destroyed after the game is initialized.
Mobs have a sentinel value preventing their A* loop from taking too long to execute. However, this means they often get partial or incomplete solutions, as in the example above. A mob (the red 8) is attempting to path to the player (red +). It visits the room it is in (white o’s) and peeks outside the room (blue o’s) and then runs out of time. The A* returns directions to an unknown, visited node.

If the walls were dynamic in nature, mobs could simply create a new map every 2 seconds when they call the A* algorithm. However, this could be potentially problematic in terms of performance.

A better way to implement the A* algorithm, which unfortunately the team did not have time to do, would have been to enqueue each adjacent node to a visit list, and then traverse only this visit list to find the path instead of traversing the entire map with each iteration. This could potentially have reduced the complexity of the algorithm from \( n^2 \) (\( n \) being proportionate to the size of the map) to some value \( k \times n \), with \( k \) being proportionate to the distance between the mob and the player (possibly \( n \times \log n \)).

In our implementation, we are using a 100x100 map grid (down from a 200x200 map grid) and our findLowestCost function is therefore iterating on the order of 800,000 times per iteration of the outer loop of the function. Instead, if we had a visit list, the findLowestCost function could iterate on the order of dozens of times. This would have a significant performance impact.

The A* function returns a vector to the mob that called it, which vector should be the direction of the next square on the map from the starting square. Because the A* is not always allowed to complete due to performance returns, mobs can sometimes have interesting behavior if they are far from the player.

Another problem with the A* that we unfortunately weren’t yet able to fix is that many walls are diagonal, and the A* can pass through these walls at a diagonal because it doesn’t see them as a solid barrier. This is a new issue that cropped up when we started compressing the map down to 100x100.
Game End Scenario

The player continues playing until their health bar drops to zero. At this time, their score is determined by the number of enemy kills they have. The game over screen is displayed and the game restarts when the player presses a key or clicks a mouse button. If an internet connection is available, their score and several other game stats are sent to a server to be stored and displayed on an online leaderboard. Their leaderboard standing is displayed on the game over screen. Additionally, a local leaderboard is stored and displayed in the terminal when the game is closed.
Conclusion

In conclusion, team First Person Strudel faced many challenges. We set ourselves an ambitious goal, and did not set out to compare our game to everyone else’s or try to determine the minimum acceptable level of effort but instead simply decided to see what we were able to do and find out how far we could go. The team pulled together remarkably well to accomplish a solid, sophisticated game, and though it is far from bug-free, it is rich in features and offers a rewarding gameplay experience.