Joe Berkheiser

Professor DeVries

CIS 612-01

24 April 2025

Brick Breaker Project Report

This final project, like all the other before it, revolved around a game. I struggled with the idea of what to do for this project for a majority of the semester. To me, it was difficult to generate requirements without some sort of outline to go off. A career centered around requirements would always have a baseline of the project from a stakeholder perspective whether their boss, or a customer. Creating a project with requirements out of the blue was a challenge for me, so I decided to make another game, and focus on the requirements for that.

One of my favorite games growing up was called Brick Breaker. I always played it on my dad’s Blackberry mobile phone. The purpose of the game was to bounce a ball, like Pong into various alignments of bricks. There were 37 levels of incrementing difficulty, and sometimes you could collect power-ups (spoiler alert: I did not add power-ups). It was the first game I can remember being really addicted to, so I decided to try and make it.

For the requirements aspect of the project, I chose to begin with a decision table. The first table I made is shown in Figure 1. I created these rules based off what I could remember from the Blackberry game and then made up some of my own. I wanted to have my basic game elements be a ball, a platform, and three different types of bricks: a broken one, a normal one, and a metal one. The user should be able to move the platform into the path of the ball to bounce it into the various brick configurations. If the ball contacts one of the bricks it would bounce off of the brick, and the brick would have its durability decreased by one. The normal bricks were given a durability of 1, the normal bricks 2, and the metal ones 100. If the user cleared all the non-metal bricks on the level, the game environment would “reset” (the platform and ball would move to their starting positions), and the level would increase. The user was given three lives which would reset for each level. Lives would be lost if the ball touched the lower boundary, below the platform. If the user ran out of lives, the game was over, and they would have to restart. I created 10 levels of increasing difficulty for the user to progress through, and if they completed each level without losing, they would win the game. This is what I attempted to explain with the decision table.

A grid of lines with different colored dots

AI-generated content may be incorrect.

**Figure 1.** Starting Decision Table.

Of course, the table in Figure 1 was a bit of a mess, so I began to condense it as best as I could. First, I decided that there were too many conditionals, so I incorporated those into their own rules, shown in Figure 2. This cleared out the majority of the conditions and made the table far more readable.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 2.** Table after one step of condensing.

Next, I noticed that Rule 9 and Rule 10 had the same result regardless of there being one non-metal block remaining. This was also true of Rule 11 and Rule 12. For both of these instances, I combined them into one rule. After that, there was not any way for me to condense the table into something that would provide more information on the project.

A white sheet with black lines and a black line

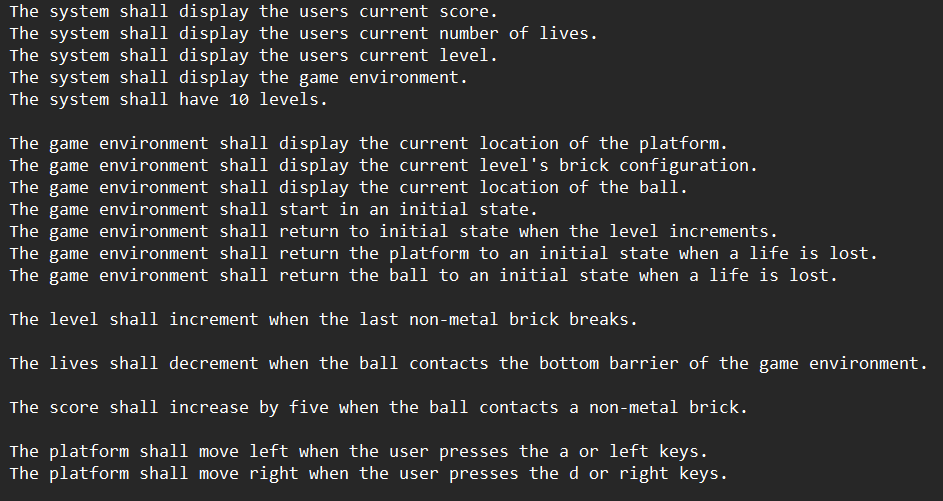
AI-generated content may be incorrect.

**Figure 3.** Table after two steps of condensing.

I think that while the table is very high-level, it was still helpful to use as a guideline for what should happen when conditions fit for certain parts of the code. Being able to check that the ball was bouncing off of the right boundary, or that four separate actions occurred correctly when the last remaining non-metal brick was broken was very helpful.

I decided to not go lower-level with the decision table because it seemed less helpful to me. Having conditions in the form of code would have just made the table less readable, and seemed unnecessary. Instead of ball colliding with right boundary, it would have been ball.posx + ball.width >= gameWidth, which without the specifics of what any of those values actually were, would be useless. Also, for each level the number of bricks and brick locations changes, so it would be supremely difficult to track all of that in a decision table.

In addition to the decision table, I made some very high-level natural language requirements for the project. These can be seen in Figure 4. I made these just to ensure that they matched what I had described in the decision table, and as a way to easily describe the high level actions that could be taken in a concise format.



**Figure 4.** Natural language requirements for the project.

Having listed some natural language requirements begs the question why did I not use all natural language requirements for this project? If the entire project was based off of natural language requirements, there would be many of them, and I do not think they would describe what actions are happening as effectively as the decision table did.

Other methods also seemed less effective. Petri nets would have been extremely large and would have been confusing very quickly. State charts would have been small, and useless since the states would have been playing, won, lost. This is also true of finite state machines. Even if I were to go more low-level, finite state machines would have little purpose for a game. For example, the ball is either colliding or not, and those states would have lots of transitions. For goal modeling, the system does not have a goal, it is just in an infinite loop of updating and rendering. I could have done a use-case diagram, but it would have been too simple, the user only interacts with the system using the arrow keys.

I coded the project following these requirements in Python, using a library called Pygame. Pygame was built in the early 2000’s to specifically for games. The coding for this project went well, but I ran into one major issue. ChatGPT defined this issue as the Tunneling Problem, and my idea of it is that when the ball moves, its moves once each loop based off of its speed. If its speed is 10, it will move 10 pixels during the loop. When a ball collides with a brick, its rarely on the same pixel, it could actually be inside of it. This can sometimes cause the ball to bounce incorrectly off of a brick, break two bricks, or actually pass through the corner of it completely. This issue was complex, and seemed to be common in game development. Currently, I am not sure how to solve it, and think of it more as a “feature” of the game.

If I have time in the future, I would like to return to this game, and improve upon what I started. This includes fixing the tunneling issue, and adding upgrades to the game. Overall, this project went far smoother than my previous game projects, and that is likely due to having knowledge of what a game-loop is, and how it works. I also think it would have been to my benefit to have done a project that would work better for more of the requirements methods we had discussed in class. Being able to do a decision table was a benefit to me because that was one of the methods that I understood the least, and getting forced into doing that due to my project was good practice for me.