Architektur Tower Defense

Da das Projekt unterschiedliche Technologien verwendet im Frontend und Backend, wurden diese in zwei separate Repositories aufgeteilt. Im ersten, das Frontend, wird alles um das Rendering des Spieles gehen. Im Backend geht es darum dem Spiel Logik zu geben und den Status zwischen den Spielern zu synchronisieren.

Da das Frontend keine Spiellogik enthält (das Spiel ist 100% Serverseitig), muss eine Architektur aufgebaut werden, welche frequente Kommunikation mit dem Backend erlaubt. Deswegen wurde entschieden, dass das Backend eine fixe Taktrate hat von entweder 30Hz oder 60Hz. Das bedeutet, dass das Backend 30- oder 60-mal pro Sekunde ein Update an das Frontend sendet. Das Frontend wird nach dem Erhalten eines Updates das SVG, welches das sichtbare Spielfeld repräsentiert, aktualisieren und dem Spieler anzeigen.

Da diese Architektur sehr schnell und zuverlässig laufen muss, wurde Rust als die Backendsprache gewählt. Es war seit längerer Zeit ein Ziel von uns Rust zu lernen, und dies erlaubt uns nicht nur eine neue Sprache zu lernen, sondern die auch gleich in einem Sinnvollen Projekt zu verwenden. Zusätzlich erhöht dies die Komplexität des Projektes, sodass es für drei Studenten geeignet ist.

Da nun Rust ausgewählt wurde, als Backendsprache muss dies mit dem Frontend, welches immer noch mit HTML, CSS, JS geschrieben wird, verbunden werden. Dafür wurde im Frontend ein asynchroner Service erstellt, welcher auf Nachrichten des Backends hört. Diese Nachrichten werden in ein Observable umgewandelt, welches von dem Game Manager synchron abgefragt werden kann. Der beschriebene Service erlaubt es zusätzlich, Nachrichten in Form von JSON Objekten an das Backend zu senden. Solche Nachrichten werden benötigt, um Spielaktionen dem Server mitzuteilen und auch den Ping zwischen den beiden Knoten festzustellen.

Im Backend läuft das ganze etwas komplizierter. Am Anfang wird ein Webserver erstellt, welcher über eine URL auf Web Socket Verbindungen wartet. Sobald eine ankommt werden zwei Threads erstellt. Der einte Thread besitzt den Game Loop dieser Verbindung und führt den 30- oder 60- Mal pro Sekunde aus. Dies wurde als separater Thread gemacht, damit mehrere Spiele gleichzeitig gespielt werden können, ansonsten könnte jeweils nur ein Spiel pro Server gespielt werden, was wenig Sinn machen würde für ein Serverspiel. Der zweite Thread ist für das zuhören von Client Nachrichten zuständig.

Damit der Game Loop die Nachrichten bekommen kann, musste jedoch eine komplizierte Brücke zwischen den beiden Threads erstellt werden. Der Grund dafür ist, dass Rust zur Kompilezeit garantiert, dass es keine Race Conditions gibt. Die erstellte Brücke funktioniert indem, dass der Client Listener ein Mutex über eine Queue übernimmt und die Nachrichten darin speichert, danach wird der Mutex wieder aufgemacht. Der Game Loop nimmt dann zum Begin jeder Iteration den Mutex und schaut ob neue Nachrichten angekommen sind und übernimmt den Besitz dieser Nachrichten, damit sie evaluiert werden können.

Da diese Architektur nun ausgebaut wurde, ist es sehr einfach das Spiel zu implementieren. Zum Begin des Spiels wird vom Backend eine Nachricht an das Frontend gesendet, welche alle Informationen über die Map hat. Darunter befindet sich z.B. die Grösse der Spielfläche und das Hintergrundbild. Danach wird das Spiel gestartet, wobei im Backend ein Ball dem Pfad der Map folgt (dieser Ball repräsentiert die Gegner, die einem Pfad folgen müssen, um an die Burg zu gelangen), die Position wird dem Frontend 30- oder 60- Mal pro Sekunden übermittelt. Das Frontend zeichnet dann den Ball mit der Position, die von dem Backend übermittelt wurde, was zu einer fliessenden Animation führt.

Schlussendlich, damit getestet werden kann, dass die Architektur mit unterschiedlichen Nachrichten auch funktioniert, wird vom Frontend jede Sekunde ein Ping geschickt. Das Backend antwortet darauf mit einer Pong Nachricht. Damit kann das Frontend die Latenz ausrechnen zwischen dem Client und Server.