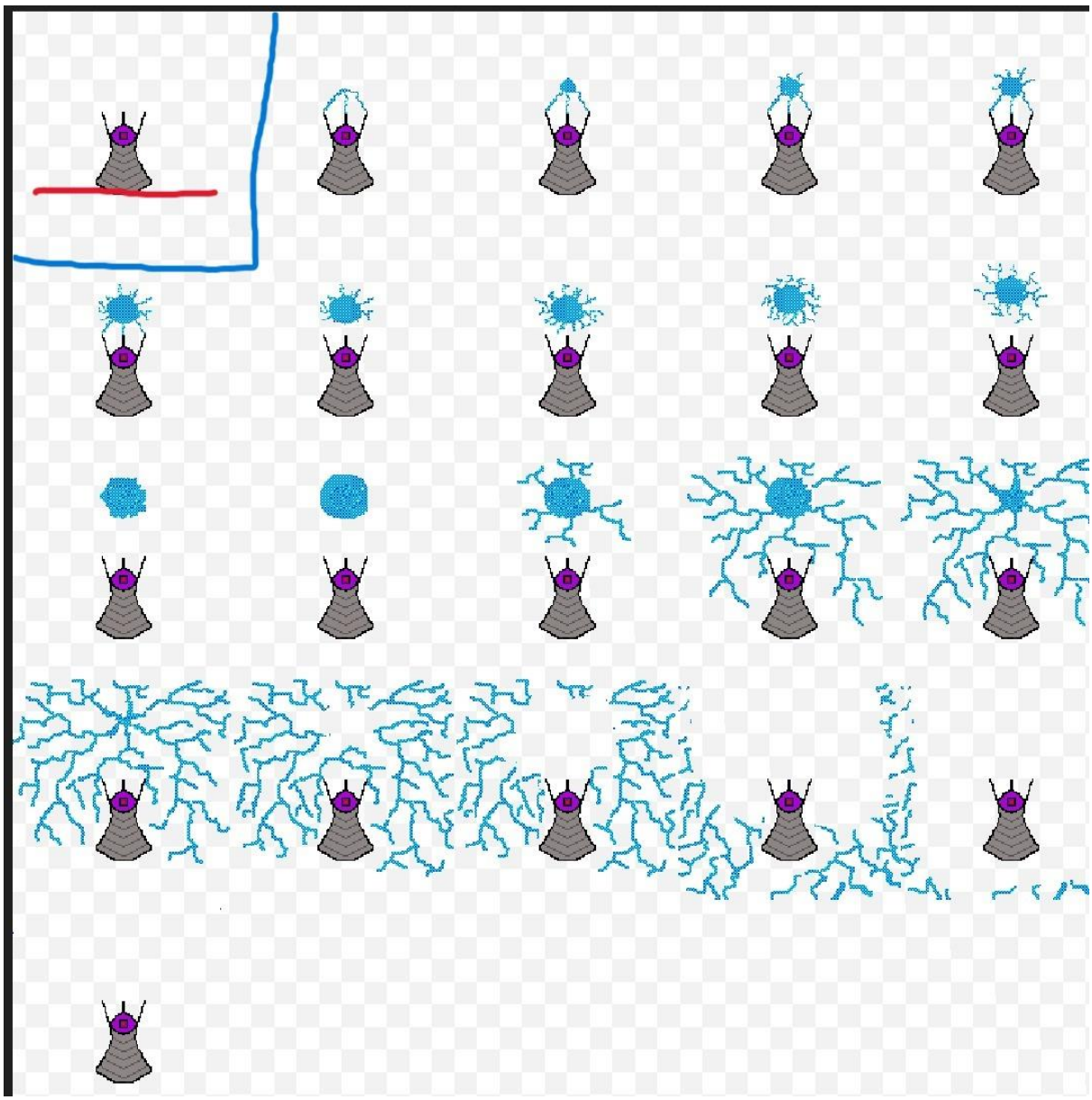
TDF-Meilenstein 2

Seit dem letzten Meilenstein ist aufgefallen, dass nach einer Zeit das Spiel sehr langsam wird. Es wurde nach ein paar Minuten so langsam, dass es nicht mehr spielbar war. Die Frames per Second, kurz FPS, sind unter 10 gefallen. Nach einer Analyze ist aufgefallen, dass die Funktion `requestAnimationFrame` exponentiell aufgerufen wurde. Das heisst, dass sie nach jedem Update vom Server und am Ende jeder Frame aufgerufen wurde.

Nachdem sichergestellt wurde, dass die `requestAnimationFrame` Methode nur nach dem ersten Update, welches das Spielfeld enthält und den Start des Spieles symbolisiert, und danach nach jedem Frame aufgerufen wird, hat sich die Performance um einiges schon verbessert. Jedoch war es immer noch zu langsam. Daher war der Fokus in diesem Meilenstein das Umbauen des Frontend von einer SVG-Architektur zu einer Canvas-Architektur. Nach dieser Umbauung der Architektur läuft das Spiel problemlos auf 144 FPS.

Eine weitere Änderung, die vorgenommen wurde, steht im Zusammenhang mit den Graphiken. Zuerst verwendete das Spiel GIFs für die Animationen von Strukturen und Gegnern. Diese wurde geändert auf Spritesheets. Dies hat zwei Vorteile. Erstens erlaubt es uns diese Spritesheets in dem Canvas direkt zu verwenden was ansonst fast unmöglich wäre zu implementieren mit GIFs. Zweitens gibt es uns die Möglichkeit die Geschwindigkeit der Animationen zu steuern. Das bedeutet, dass wir zum Beispiel mit diesem neuen Ansatz Slow Motion Animationen unterstützen können.

Zusätzlich wurde klar, dass die existierenden Graphiken neu überarbeitet werden müssen. Zuerst gab es Animationen die Unter einer Struktur waren. Ein Beispiel dafür gibt es auf folgender Graphik. Das blau gezeichnete Rechteck zeigt ein Frame des Spritesheets. Dabei ist ersichtlich, dass der Turm in der Mitte liegt und die Animation des Turms den ganzen Rest aufnehmen kann. Bei dem Spielen ist dabei aufgefallen, dass der Turm nicht dahin platziert wird, wo man es erwartet. Daher wurde eine neue Regel erstellt, die Strukturen müssen an der rot gezeichneten Linie ausgerichtet werden. Somit kann sichergestellt werden, dass der Turm da platziert wird, wo der Spieler ihn auch erwartet.



Schlussendlich wurde aufgefallen, dass sehr viele unnötige Daten pro Update von dem Server an den Client gesendet wird. Da der Server 30-mal ein Update pro Sekunde dem Client sendet, sollten keine unnötige Felder mitgesendet werden. In unserem Fall gibt es pro Struktur dynamische und statische Daten. Dynamische Daten sind solche, die pro Instanz unterschiedlich sind. Statisch Daten sind alle anderen die immer dieselben sind. Nehmen wir als Beispiel den Turm «Grunt» (Name ist noch nicht definitiv) dieser hat eine Position und Leben. Jede Instanz von diesem Turm hat eine eigene Position und wie viel Leben er noch hat, daher müssen diese Attribute für jede Instanz separat gespeichert werden. Der Turm hat jedoch auch noch eine Referenz auf das Spritesheet, welches immer dasselbe sein wird für jede Instanz. Um diese überflüssigen Daten zu minimieren, wurde das Flyweight Pattern implementiert. [1]

In dem Flyweight Pattern geht es darum die statischen von den Dynamischen Daten zu separieren. In der folgenden Graphik gibt es ein Beispiel von Instanzen, welche dieselben statischen Datenfeldern besitzen, somit wird wertvoller Speicherplatz verloren gegangen.

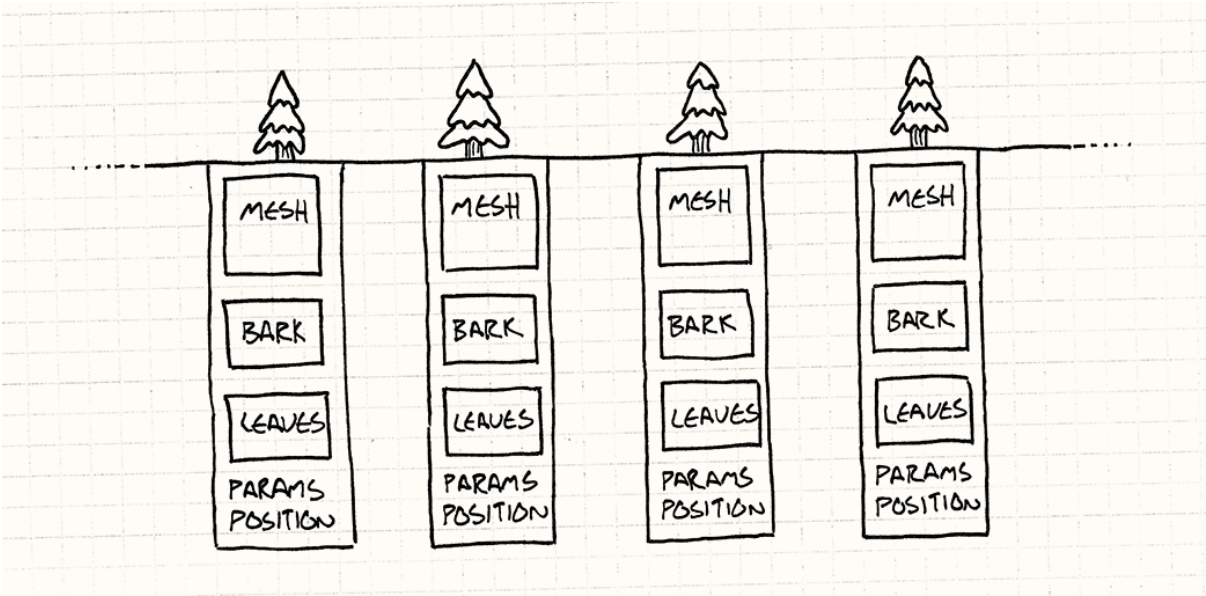


Abbildung 1 - Instanzen bevor der Implementierung des Flyweight Pattern

Nachdem das Flyweight Pattern eingesetzt wird, werden alle statischen Felder in eine geteilte «Modell» Klassen geladen. Jede Instanz kriegt dann eine Referenz auf diese geteilte Modellklasse. In der folgenden Graphik sind dieselben Daten ersichtlich, jedoch optimiert durch das Pattern.

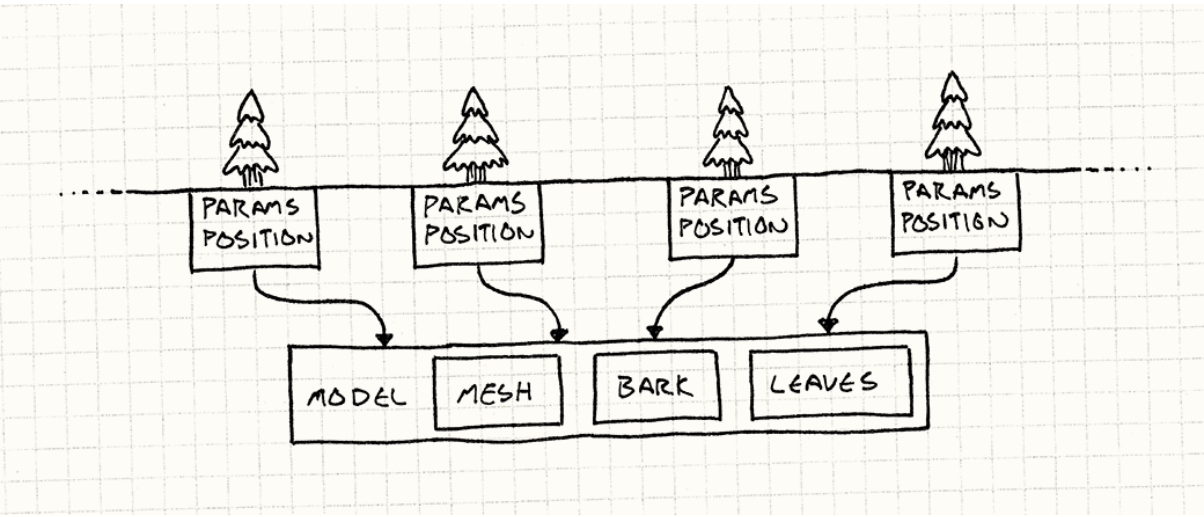


Abbildung 2 - Instanzen nach der Implementierung des Flyweight Pattern

Referenzen

[1] N. Robert, "Flyweight," in *Game Programming Patterns*, Genever Benning, 2014.