Performance Tipps

- Einführung
  - Overdraw verringern
  - Effiziente Nutzung des Canvas
    - Nested Canvasses
    - Graphics Raycaster
  - Layouts
  - Der Animator

Einführung

Um bessere Performance aus der UI herauszuholen müssen wir uns vor allem zweier Dinge bewusst sein:

1. Sämtliche UI ist transparent.
2. Wie funktionieren Canvas und Layouts oder genauer gesagt, wann müssen sie erneuert werden?

Bevor es jetzt weiter geht noch ein **Hinweis**: Wie bei allem was mit Performance zu tun hat, gibt es kaum einen universell besten Weg Dinge anzugehen, sondern spezifische Lösungen für Probleme mit der eigenen UI. Diese Lösungen können dann wieder an anderen Stellen Dinge kaputt machen, beispielsweise sind mehrere Canvasses eine gute Idee, jedoch erhöhen sie auch die Anzahl an Batches; Spritemesh wird Overdraw verringern, allerdings gibt es jetzt mehr Polygone und man muss die Importeinstellung "Tight" verwenden. Es ist also ein Ausprobieren und Abschätzen, was für einen den meisten Mehrwert bringt. Am Ende kann wahrscheinlich nur der Profiler wirklich bei der Entscheidung helfen.

Quelle: Unity Webinar

Overdraw verringern

Gerade auf Mobile ist Overdraw ein Performance-Killer. Er entsteht, wenn sehr viele transparente Objekte über einander liegen. Es werden also oft Pixel gerendered, die später im Rendervorgang überschrieben werden, weil sie tatsächlich verdeckt sind. Das trifft auf die Gesamte UI und die meisten Sprites zu.

Unity bietet Möglichkeiten an Overdraw zu visualisieren, welche es gibt ist abhängig von der verwendeten Renderpipeline. Der Modus für die Built-in-Renderpipeline sieht beispielsweise so aus:

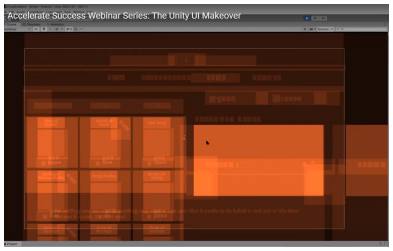

Abbildung 1: Der Overdraw Shader färbt stellen mit überlagernder Transparenz heller ein

Es ist also allgemein ratsam beim Bau von Elementen darauf zu achten, dass alle statischen Elemente zu einer Textur zusammengeführt werden. Ein Bereich in Abbildung 1 mit sehr viel Overdraw sind die Warenangebote auf der linken Seite. Eines davon soll im Folgenden als Beispiel dienen:

Abbildung 2: Ein Warenangebot aus einem prototypischen Shop

Abbildung 3: Die Grafik besteht aus sehr vielen Einzelteilen

In den Abbildungen 2/3 kann man sehen, dass die unscheinbare Karte tatsächlich aus vielen Einzelteilen besteht. Diese Einzelteile liegen übereinander und werden trotzdem alle voll gezeichnet, bevor sie von dem nachfolgenden Element übergemalt werden.

Abbildung 4: Ein Rahmen mit transparenten Pixeln in der Mitte

Abbildung 5: Der Bogen ist sehr unvorteilhaft rotiert

Wenn wir dann noch die Rahmen für Ware und Namen gesondert betrachten sehen wir viele transparente, also leere, Pixel auf den Texturen. Tatsächlich kosten uns diese leeren Pixel trotzdem Performance, weil sie erst beim Rendern aussortiert werden. Noch schlimmer ist es beim Bogen in Abbildung 5, dessen Rotation die Textur unnötig mit Leerraum füllt. Man sollte also bereits beim Export von Grafiken darauf achten, dass sie möglichst wenig leere Pixel enthalten und auch nicht unvorteilhaft rotiert sind.

Abbildung 6: Zusammengefügter Hintergrund für die Ware

Abbildung 6 zeigt, wie eine bessere Version aussehen würde. Alle Einzelteile sind zu einer einzelnen Textur reduziert worden. Um der schlechten Rotation des Bogens Herr zu werden, gibt es in Unity noch die Option "Use Sprite Mesh". Dadurch wird für den Sprite ein mehr oder minder genaues Mesh erstellt, anstatt die Textur auf einem Quad aufzuspannen.

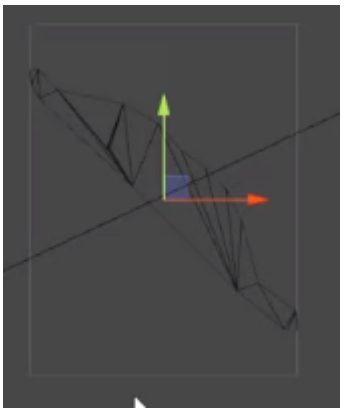

Abbildung 7: Wireframe-Ansicht; Der Bogen aus Abbildung 5 ist nun ein Mesh

Durch diese Einstellung erzeugt der Bogen weniger Overdraw, allerdings kann man in Abbildun 7 auch sehen, dass er nun mehrere Dreiecke braucht, während es vorher nur zwei waren. Es wird ebenfalls die Einstellung "Mesh Type - Tight" beim Import benötigt.

Quelle: Unite 2017
Quelle: Unity Learn
Quelle: Unity How To

Effiziente Nutzung des Canvas

Allgemein wird der Inhalt von Canvasses über ein generiertes dynamisches Mesh gerendered. Dieses Mesh wird jedes mal neu generiert wenn sich etwas innerhalb des Canvas verändert. **Wenn sich also ein einzelnes Element verändert, muss das ganze Canvas neu generiert werden.** Zusätzlich dazu muss auch noch das Layout neu generiert werden - warum das ebenfalls schlecht ist kommt im Abschnitt Layouts.

Eine erste Faustregel ist es also mehrere Canvasses zu benutzen und nicht die gesamte UI in ein riesiges Canvas zu Packen.

Im nächsten Schritt kann man dann ein jedes Canvas noch unterteilen, indem man Sub - bzw. Nested Canvasses hinzufügt.

Nested Canvasses

Die grundsätzliche Idee ist es UI in statische bzw. dynamische Elemente und damit auch Canvasse zu unterteilen. Dadurch sollen Elemente die sich manchmal bis sehr häufig verändern von denen, die das gar nicht tun, abgeschottet werden, damit sie nicht in Neu-Berechnungen eingeschlossen werden.

Im Artikel von Unity geht es um einen Timer. Dieser besteht aus einem Text und einer Zeitanzeige. Es ist klar, dass sich der Text "Current time" nie ändern wird, während die Anzeige der Zeit sich kontinuierlich updated.

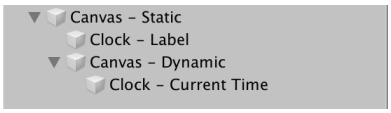

Abbildung 8: Nested Canvas Struktur des beschriebenen Timers

Entsprechend Abbildung 8 bringen wird die Zeitanzeige in ein eigenes Subcanvas, unter dem Canvas auf dem der Text liegt. Wenn sich jetzt die Zeit verändert, wird nur das Subcanvas neu berechnet - der Text bleibt unberührt.

Graphics Raycaster

Den Graphics Raycaster benötigt man auf jedem Canvas (und Sub-Canvas!) das (Touch-)Input erhalten soll. Tatsächlich ist er nicht wirklich ein Raycaster, sondern prüft ob sich ein Punkt innerhalb eines Rechtecks befindet und das für jedes RectTransform unter dem Canvas, dass als interaktiv markiert ist.

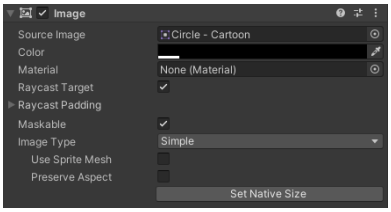

Abbildung 9: Interaktive Image Komponente

Als interaktiv markiert sind alle Komponenten mit dem Toggle "Raycast Target" auf an (unter anderem Image; siehe Abbildung 9).
Faustregel: Achte darauf den Toggle auszuschalten, wo er nicht benötigt wird! ... oder entferne den Raycaster.

Quelle: Unite 2017

Layouts

Auto Layout funktioniert über ein sog. "dirty flag"-System. Wenn sich ein Layoutelement verändert und es damit einen umgebenden Layoutcontroller ungültig macht (z.B. Size oder Scale verändert), wird es als "dirty" markiert, worauf das Layoutsystem dann reagieren kann.

Problem: Layoutelemente sind Komponenten, also kann auf jedem Element oder auch Parent eins oder mehrere sein.

Um die Neu-Berechnung des Layouts korrekt auszuführen, wird nach dem Layoutcontroller gesucht, der am weitesten oben in der Hierarchie steht. Das ganze passiert natürlich über GetComponent() auf jedem Objekt. So wird jedes Element, dass sein Layout auf dirty setzen will, minimal einen Aufruf von GetComponent() nach sich ziehen. Jeder geschaltete Layoutcontroller vervielfältigt dieses Problem.

Wodurch wird ein Layout als dirty markiert?

Kurzform: **Durch fast alles ...**

Nur ein paar Beispiele:

- **OnEnable() und OnDisable(),**
- **Reparenting,**
  Doppelt; einmal für den alten und neuen Parent
- **OnDidApplyAnimationProperties(),**
  Wenn ein Animator mit dem Element interagiert
- **OnRectTransformDimensionsChanged()**
  z.B. Skalieren oder Resizing, Änderung von Position oder Kamera

Lösungen: Was kann man dagegen tun?

- **Vermeidet Layoutelemente wo es geht,**
  Oft genug reicht es nur mit festen oder relativen Ankern zu arbeiten.
- **Deaktiviert Layoutelemente nachdem sie ihre Arbeit getan haben,**
  Viele Layouts müssen sich nur einmal nach dem Laden des Spiels aufbauen und verändern sich dann nicht mehr. Das gilt insbesondere für statische Anzeigen. Auf all diesen Objekten kann man die Layoutelemente deaktivieren, um sich das dirtying zu sparen.
- **Deaktiviert die Canvas Komponente an Stelle des GameObjects, wenn ihr ein ganzes Canvas ausblenden wollt,**
  Dadurch wird nichts mehr angezeigt, aber man spart sich OnDisable() und OnEnable() + dirtying.
- **Suche nach zuständigem Layoutcontroller über leeres Objekt unterbrechen,**
  Die Suche nach oben wird beendet, sobald kein Layoutcontroller mehr gefunden wird. Das ist vor allem hilfreich, wenn man Objekte über eine Animation skalieren will, dies aber nicht das Layout beeinflussen soll (z.B. Button drücken mit leichtem bounce).

- **Disablen vor Reparenting,**
  Beispielsweise für Pooling von UI Elementen. Zuerst disablen, dann zurück in den Pool. Beim Nehmen aus dem Pool zuerst Reparenting, dann enablen. Das Ganze dient dazu, die Anzahl an no-op dirty Calls zu maximieren.
- **Eigenen Layout-Code schreiben, der on-demand updatet.**
  Wenn sowieso viele Änderungen an einem Element im gleichen Frame anfallen wäre es besser erst alles zu updaten, wenn alle Änderungen durch sind. Das kann Unity natürlich nicht von allein.

Quelle: Unite 2017

Der Animator

Kurz gesagt: Benutzt keinen Animator für die UI. Stattdessen solltet ihr Code und/oder Tweens verwenden um eure UI zu animieren. Für Tweens gibt es das sehr gute Package DotweenPro.

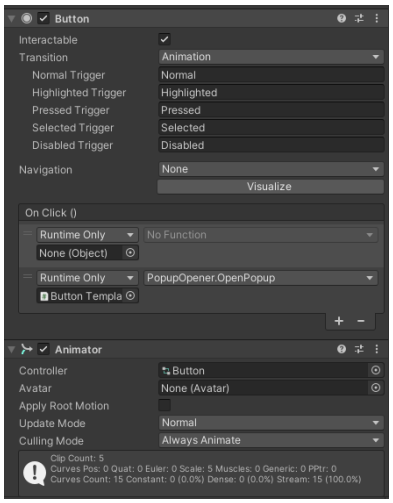

Abbildung 10: Unity Standard Button

Warum ist ein Animator Setup, wie in Abbildung 10, schlecht?

Ein Animator wendet jeden Frame seine Werte auf die animierten Objekte an, egal ob sich Werte in der Animation verändert haben oder nicht. Das bedeutet, dass jeden Frame alle animierten Objekte als dirty markiert werden, was diverse Aktionen im Layout bzw. Canvas Code nach sich zieht.

Entsprechend ist der Animator nicht dafür geeignet States abzubilden - wie z. B. "highlighted, selected" etc. im Beispiel - sondern sollte nur zum Abspielen von Animationen verwendet werden, da sich in diesem meist sowieso jeden Frame etwas verändert, was dann eh dirtying verursacht.