Setup Guide und Infos

Inhalt

[2 Git 2](#_Toc525916800)

[2.1 Grundsätzlicher Ablauf: 2](#_Toc525916801)

[2.1.1 Workflow (merge) 2](#_Toc525916802)

[2.1.2 Workflow (mit rebase): 3](#_Toc525916803)

[2.2 Issues/Todos 4](#_Toc525916804)

[2.3 Commits 4](#_Toc525916805)

[2.4 Other 4](#_Toc525916806)

[3 Unity und Coding 4](#_Toc525916807)

[3.1 Architektur, tips, best practices 5](#_Toc525916808)

[3.1.1 Generelle Tipps 5](#_Toc525916809)

[3.1.2 Hilfreiche Shortcuts 9](#_Toc525916810)

[3.1.3 Scriptable Objects 10](#_Toc525916811)

[3.2 Assetpacks 10](#_Toc525916812)

[3.2.1 PackageManager 10](#_Toc525916813)

[3.2.2 AssetStore 10](#_Toc525916814)

[3.2.3 Maybe/Later 11](#_Toc525916815)

*Bitte einfach neue Sachen dazuschreiben, ergänzen und ändern falls ihr was habt*

# Git

Für git verwenden wir Featurebranches und Mergerequests. <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>

TLDR: jeder arbeitet auf seinen eigenen Branches (z.B. Feature/meingeilesfeature) und wenn das Feature fertig ist merged man das in den Master-Branch.

Mit gitflow geht das mit Featurebranches erstellen nochn bisschen schneller, aber sonst einfach immer ne neue Branch erstellen. <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

Wir verwenden keinen Develop branch sondern nur Master.

Mit einem git GUI ([sourcetree](https://www.sourcetreeapp.com/), smartgit, tortoisegit, sublimemerge etc.) ist Änderungen und Konflikte erkennen etwas übersichtlicher und vielleicht einfacher.

<https://git-scm.com/download/gui/windows>

## Grundsätzlicher Ablauf:

*Neuen Featurebranch erstellen -> arbeiten -> Feature fertig. Dann neuesten Stand von master pullen und master in die eigene branch mergen/rebasen. (am besten zwischendurch immer wieder den master reinmergen) Dann auf gitlab einen mergerequest erstellen. Dazu auf Merge Requests klicken -> new mergerequest und eure branch als source und master als target auswählen.   
Den mergerequest kann dann jeder von uns mergen.*

Immer wenn ein neuer Mergerequest erstellt wird bekommen wir eine Nachricht in unserem Discord channel.

**Merge vs. Rebase**

<https://www.atlassian.com/git/tutorials/merging-vs-rebasing>

### Workflow (merge)

git checkout master -> auf master wechseln  
git pull -> neuesten Stand holen  
git checkout -b #branchname# -> neuen Branch erstellen  
*arbeiten*  
git push -u origin #branchname# -> branch auf gitlab pushen, erstes Mal mit -u  
*fertig arbeiten*  
git merge master -> master in eigene branch mergen  
*eventuelle konflikte resolven*

Gitlab Mergerequest:   
*auf eigene Branch gehen -> mergerequest erstellen  
Eventuell einen Assignee auswählen  
Sourcebranch -> eigene Branch  
Targetbranch -> normal master, oder so man halt reinmergen will  
Je nachdem ob die Branch am remote bleiben das ein Häckchen setzen*

### **Workflow (mit rebase)**:

Immer zu erst alle Änderungen vom master holen, um auf dem neuesten Stand zu sein.

Git checkout master 🡪 aktuellen Branch auf master ändern

Git pull --rebase 🡪 alle Änderungen am master vom Server herunterladen

**Neuen Branch erstellen / Mit Branch weiterarbeiten**

Git checkout -b BranchName 🡪 Neuen Branch für neues Feature erstellen

**Oder**: Git checkout BranchName 🡪 Auf bereits bestehenden Branch wechseln

Git rebase master 🡪 Alle Änderungen vom Master auf den aktuellen Branch anwenden

Hier kann es zu Merge Konflikten kommen, da die Version deines Branches mit der Version des Masters im Konflikt stehen kann

**Falls Merge Konflikt:**

Git status 🡪 um schnell zu erkennen, in welchem File ein Konflikt vorliegt

Konflikt beheben (entweder Version vom Head oder eigene Version nehmen)

Git add files

Git rebase --continue 🡪 Um mit dem Rebase fortzufahren, hier kann erneut ein Konflikt auftreten, einfach wie vorher beheben, adden und rebase continue

Wenn alle Konflikte behoben sind, kann man ganz normal mit dem Branch auf der aktuellsten Version weiterarbeiten.

**Alles erledigt:**

Ist alles erledigt, ganz normal git add files, git commit

Git push -u origin BranchName 🡪 alle Änderungen auf den Server laden (-u origin nur beim ersten Mal) , in der Konsole steht gleich der Link zum Merge Request dabei (Strg – Klick)

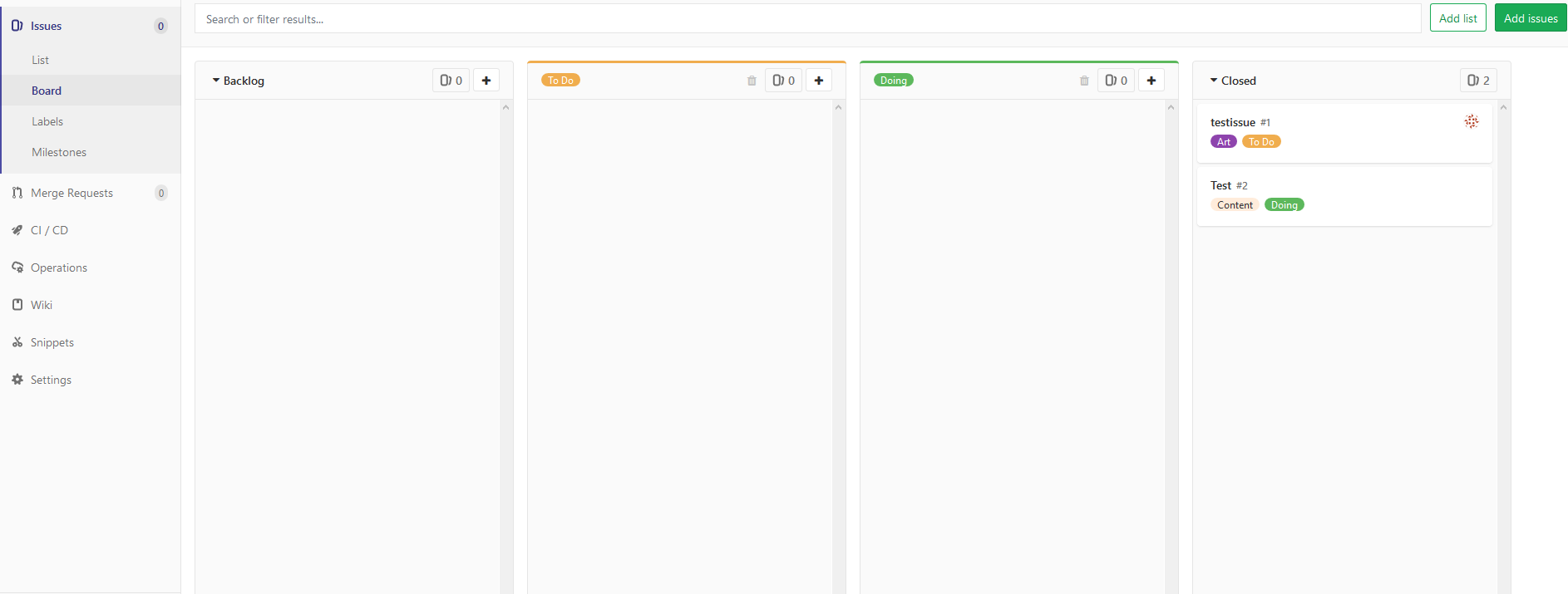
**Merge Request auf Github erstellen**

Zielbranch auswählen (master), Assignee auswählen (nicht selbst), Häkchen bei „remove source branch when merge request is accepted.“

Submit -> Jetzt kann jemand anderes den Merge Request reviewn und den Merge bestätigen. Der master branch enthält nun die Änderungen von deinem Branch.

## Issues/Todos

Auf gitlab gibt es ein eigenes Board mit Issues/Todos.



Auf discord gibs auch eine Notification im Channel wenn Issues erstellt/geclosed werden. Man kann auch Leuten issues zuweisen die kriegen dann auf Gitlab rechts oben eine Benrachrichtigung und per Email auch (Email kann man unter Profilsettings/notifications ausstellen).

## Commits

Wenn wir ein Assetpack hinzufügen bitte nur das hochpushen was wir wirklich brauchen. Also am besten erstmal lokal herumprobieren und dann große Exampleszenen etc. rausnehmen und nicht hochpushen.

<http://karma-runner.github.io/2.0/dev/git-commit-msg.html>  
Wir verwenden den Style für commit messages bei polycular, müssen wir uns noch ausmachen ob/was wir da hernehmen.

## Other

Sonst am besten nachn clonen nochmal git lfs install übers Repo rennen lassen damit das auch fix funktioniert.

Und develop auschecken und mit dem als base arbeiten

# Unity und Coding

* Version: 2018.3.x
* Unser Spiel liegt im Ordner \_Game. Das ist, weil alle Assetpacks die man importiert keinen einheitlichen Ordner haben. So ist unser Spiel getrennt von den 3rd party assetpacks.
* Scripts bitte in den Scripts Ordner, am besten dort mit Rechtsklick erstellen und nicht direkt am Gameobject dann muss man die nicht immer moven.
* Eigene Szenen unter Scenes/Testbed
* Die Placeholder.txt sind da weil git keine leeren Ordner pushed, also die kann man löschen sobald was im Ordner ist.
* Unter preferences -> cache server könnt ihr einen lokalen Cacheserver einstellen, falls ihr irgendwie Platformen wechselt oder irgendwas reimportiert was schonmal drin war braucht ihr dann nicht ewig wieder importieren.
* Aufm discord ist ein neues defaultscript für Unity, einfach mit unseren Namen als Credits und Namespace. Das könnt ihr unter #unitypath#/editor/data/resources/scripttemplates einfügen. <https://cdn.discordapp.com/attachments/491343220895383552/493414804980236300/81-C_Script-NewBehaviourScript.cs.txt>
* <https://csharpcodingguidelines.com/naming-guidelines/> Coding guidelines, Resharper->Options->code editing->C# naming style anpassen.
* <https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-6> neue C# 6 und 7 features sind vllt ganz nützlich
* Neue Prefabs in 2018.3! <https://www.youtube.com/watch?v=6_gAiPPKyUg>
* Resolution Target 1920x1080

## Architektur, tips, best practices

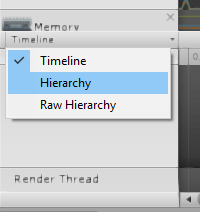
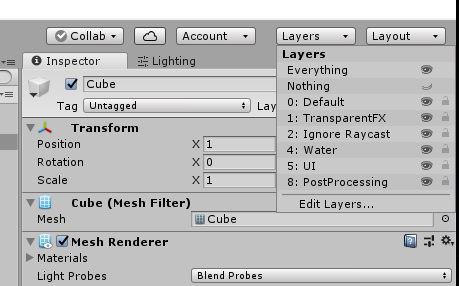
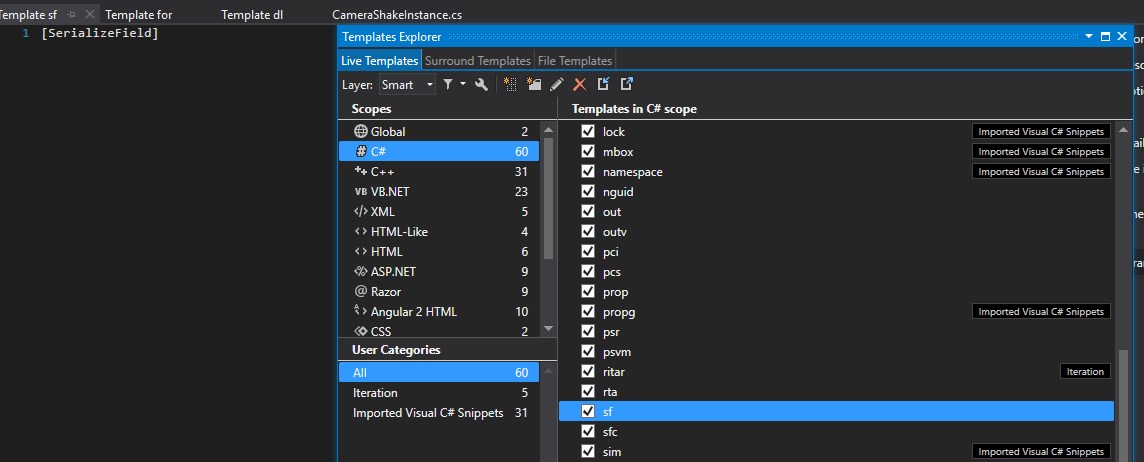
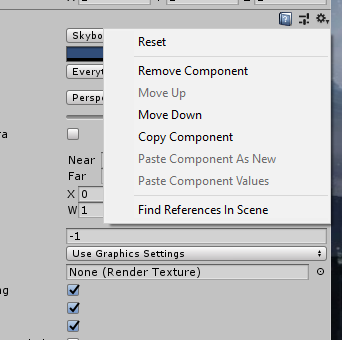
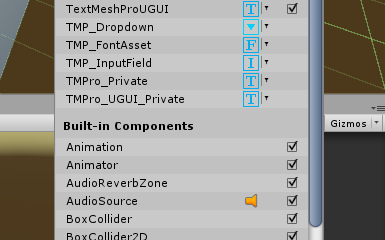
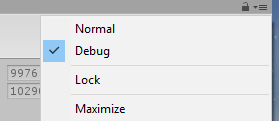
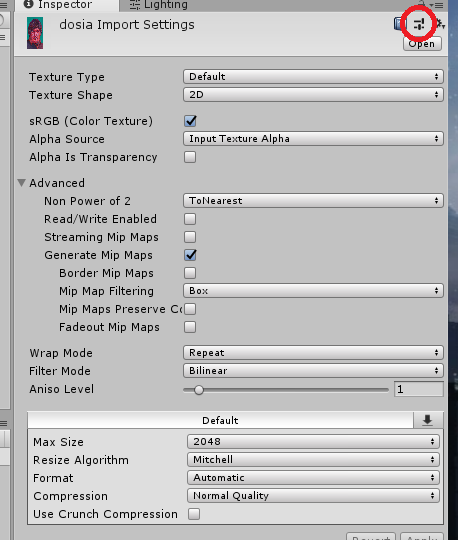
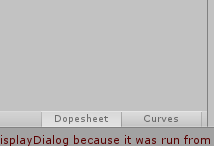
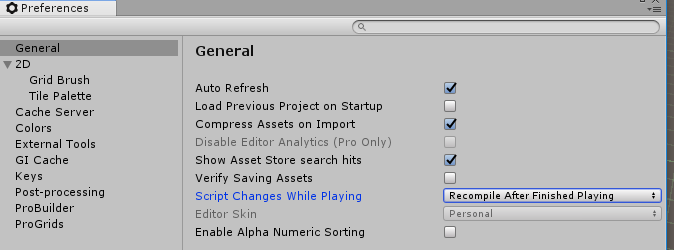
<https://unity3d.com/de/learn/tutorials/s/best-practices>

<https://trello.com/b/Z6cDRyis/good-coding-practices-in-unity>

Das Trello board is ziemlich interessant zum durchlesen und auch ned zu lang

<https://unity3d.com/de/learn/tutorials/s/tips>

### Generelle Tipps

* Immer kleine Scripte machen die nur eine Funktion haben und am besten für sich selbst funktionieren.
* Kein GetComponent<comp> im Update loop; abc.transform ist das gleiche wie abc.GetComponent<Transform>
* Öfter mal den Profiler benutzen und das Game länger laufen lassen damit man früh auf Performanceprobleme draufkommt.
* Der Profiler hat verschiedene Ansichten, Timeline ist ganz praktisch 
* Für Physics, Rendering und allgemein eigene Layer verwenden. Unter settings/physics kann man dann auswählen was mit was kollidiert -> performance.
* Layer kann man separat ausblenden/nicht anklickbar machen. 
* Objekte und Szenen so aufbauen dass sie einen realistischen Scale haben (1 unity unit = 1 meter)
* Scale von Parentobjekten am besten auf 1 lassen, vorallem bei UI elementen den scale auf 1 lassen, kann sonst zu ziemlich komischen Verhalten führen.
* Dasselbe gilt für 3D Modelle, die sollten auch einen einheitlichen Scale haben
* [SerializeField] statt public Variablen verwenden, finde ich generell besser damit nicht was public ist was nicht public sein muss.
* Unter Resharer -> Tools -> Template explorer könnt ihr für SerializeField, Debug.Log usw shortcuts erstellen, dann braucht ihr z.B. nur sf für [SerializeField] tippen. 
* Mit Attach to unity in visual studio kann man breakpoints setzen und debuggen mit Unity.
* Rechtsklick -> copy component um Änderungen ausm Playmode beizubehalten 
* „Find References in Scene“ hier oder auf ein Asset im Ordner um alle in der Szene zu zeigen.
* [Header(„abc“)] und [Tooltip] etc. verwenden um Variablen im Inspektor schön zu gliedern und zu erklären.
* Für wichtige Komponenten zahlt es sich vielleicht aus einen custom Inspector zu machen falls es viel verwendet wird <https://www.youtube.com/watch?v=RInUu1_8aGw> oder alternativ Property drawers <https://docs.unity3d.com/Manual/editor-PropertyDrawers.html>
* Störende Gizmos kann man ausblenden 
* Im debug view auf komponenten kann man schnell nachschauen was für Werte z.B. private variablen haben die nicht im Inspector angezeigt werden 
* Man kann import settings als presets speichern 
* Inspector locken um z.B. leichter Variablen zu assignen 
* Es gibt transform.SetPositionAndRotation, anstatt transform.position und rotation einzeln zu assignen
* Wenns geht und sinnvoll ist structs statt Klassen verwenden (keine allocation, Vorteil z.B. wenn man schnell viele objekte erstellt)
* Im AnimationWindow die Kurven verwenden statt die Values zu editieren 
* Für Objekte mit Rigidbodies (nicht kinematic) am besten rigidbody.MoveRigidbody verwenden statt transform.Translate o.Ä.
* Resharper hat eine Unity extension. Resharper->ExtensionManager suchen
* C# Extension methods <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods>
* Ihr könnt einstellen ob Scripte kompilieren während ihr im Playmode seid 
* Für oft verwendete Strings ist es vielleicht gscheid eine statische Klasse zu machen wo alle strings definiert sind oder alternativ Scritptable Object als Datacontainer
* In den scripts checken ob die variablen assigned oder null sind und dann fehlermeldungen ausgeben
* Verschiedene UI Canvas verwenden für statische/dynamische UI Elemente
* Assertions verwenden <https://docs.unity3d.com/ScriptReference/Assertions.Assert.html>
* [RequireComponent] für Scripte die Abhängigkeiten haben und [DisallowMultipleComponent]
* [Range(1,2)] für Variablen die ein max/min Bereich haben sollen
* Mathf.Approximately = float comparer
* Lerpen: <https://chicounity3d.wordpress.com/2014/05/23/how-to-lerp-like-a-pro/>
* Warnings immer versuchen zu fixen, damit nicht später 100 verschiedene dastehen und man eine wichtige übersieht

### Hilfreiche Shortcuts

Strg+shift+n: new empty gameobject

Shift+alt+n: new empty go als child

Shift+Space: Ausgewähltes window maximieren

Man kann selections saven: <https://twitter.com/Twirlbound/status/1041987944133865472>

Gameobject/Move and align with view, gut für Cameras



Alt+left arrow/right arrow: Die Children eines Objektes komplett ein/ausklappen

2x F oder shift+f um einem Objekt zu folgen

### **Scriptable Objects**

können ziemlich hilfreich sein, besonders für UI aber auch für andere Sachen.

Einfach ein Scriptable Objekt als Profile/Setting definieren und im Inspector zuweisen, ziemlich praktisch und man hat keine Probleme mit falsch geschriebenen Strings etc.

<https://www.youtube.com/watch?v=HkUSmI7F304> <- der UI Talk ist nicht zu lang und ziemlich nice

<https://www.youtube.com/watch?v=aPXvoWVabPY> <- 10min Video von Bracheys

<https://blogs.unity3d.com/2017/11/20/making-cool-stuff-with-scriptableobjects/> <- in dem Artikel gibs auch nochmal cool Erklärungen, die Talks darin sind auch gut aber ziemlich lang (beide 1h).

## Assetpacks

Hier eine Übersicht über die Assetpacks die ich schonmal eingemacht hab mit Erklärungen dazu.  
Derweil nur die essential Assets.

### PackageManager

**Post Processing**   
<https://github.com/Unity-Technologies/PostProcessing/wiki/Quick-start>

Wenn wir Szenen bauen am besten immer vorher auf die Kamera Post Processing tun damit es ungefähr so ausschaut wie es dann am Ende soll.

**Probuilder**   
<https://www.youtube.com/watch?v=PUSOg5YEflM>

zum easy Prototypen von Objekten

**ProGrids**<https://www.youtube.com/watch?v=UtNvtIrJcNc>

**Textmesh Pro**

Bessere und performantere Texte, am besten immer verwenden für alle Texte im Spiel.

**Vector Graphics**

Support für Vectorgrafiken

### AssetStore

**Easy Notes**

Notizen unter Window/Easy Notes für Projekt

Szenennotizen auf Gameobjects indem man das Easynote script hinzufügt.

Ist denke ich ganz praktisch.

**EZ Camera Shake**

Camerashake - ziemlich wichtig fürs Gamefeel

### Maybe/Later

**Aura**https://assetstore.unity.com/packages/vfx/shaders/aura-volumetric-lighting-111664

Volumetrisches Lighting

Da ist ein großer Computeshader dabei der ziemlich lange zum importieren braucht, einfach laufen lassen, dauert ~30min-1h.

**Gaia**https://assetstore.unity.com/packages/tools/terrain/gaia-42618

Terrain Tools

**Unity Particle Pack**

<https://assetstore.unity.com/packages/essentials/tutorial-projects/unity-particle-pack-127325>

**Unity UI Extensions**

<https://bitbucket.org/UnityUIExtensions/unity-ui-extensions>

**Photo Bolt**

Als Networking Solution

<https://assetstore.unity.com/packages/tools/network/photon-bolt-free-127156>

**Utility Extension Methods**

<https://gist.github.com/omgwtfgames/f917ca28581761b8100f>

**Free Script Collection List**

<https://github.com/michidk/Unity-Script-Collection>