# Verwendete Ressourcen

* Soundeffekte (Plop, Explosion ...):   
  <https://pixabay.com/sound-effects/search/?order=ec>
* Icons für schnelles Prototyping, Beispiel ‚bycicle repair‘  
  <https://www.flaticon.com/search?word=bicycle%20repair>

# Iterationen und Play-Testing

* Ist das Spielziel verständlich?
* Wie wirken Grafik, Sound und Gameplay?

Was hat sich schon verändert?

* Zielparameter (d.h. Spielregeln)
  + Aktuelle Ziel-Parameter: Lives, Speed und Veggie Points
  + Ursprüngliche Ziel-Parameter: ‚skills‘, ‚strength‘ und ‚lives‘
  + Das sich die Spielentwicklung an einer speziellen Spielmechanik orientiert ist nicht ungewöhnlich . (Mechanik => Ästhetik => Story). Je nach Stärken im Team ...
* Ästhetik
  + Welche Icons machen Sinn ... siehe Ordner ‚Spare\_Icons‘
  + Zufällige Rotationen einsetzen (gleichmäßig ausgerichtete drops sehen strange aus)

# ‚World‘ Scene

* **Reuse:** Playermovement und Terrains (Autotiling) wurden aus ‚06\_Sproutworld‘ übernommen (Aussehen kann per Tilemap und Terrains jederzeit verändert werden)
* **Größenverhältnisse**: Gamefläche ist ≈ 1100 x 620 pixel / Camera Zoom = 5 in der ‚Player‘ Scene / Display-Window ist 2260 x 1240 (richtet sich nach der Auflösung des Rechners)

*DisplayServer.window\_set\_size (Vector2(2260, 1240))*

# ‚Player‘ Scene

* Die meisten Signale werden diesmal vom Characterbody2D getriggert, da dieser sein Verhalten entsprechend der Kollisionen ändert. Wichtig ist wieder, die Struktur des Signals (d.h. die Bestandteile) im Auge zu behalten. Die Playerfigur kann nur triggern, wenn sie ein Area2D hat und auf eine andere Area2D oder einen anderen Characterbody2D (z.B. Enemy NPC) trifft.
  + - Typische Player Knoten Struktur

Ein Bild, das Text, Schrift, Screenshot enthält.

Automatisch generierte Beschreibung

* Neues Pixel Artset ‚Cave Theme‘ ist jetzt in Github im Ordner ‚Asset\_Packs‘ zu finden (Internetquellen sollten immer in einem .txt File sein.
* Am Ende eines Spieles könnte man die Credits wie Film durchlaufen lassen (es gibt schon eine rudimentäre ‚Credits\_Scene‘ ), die dann in game\_ui.gd getriggert wird. Mehr Info im Abschnitt “**Tweens und die ‚credits\_scene‘ “**

# Automatische ‚drops‘ / Item Generierung

* Quellen:

Random positioning <https://www.youtube.com/watch?v=VSrsoeufgvo>

Random numbering <https://youtu.be/UNb0Gise_jA?si=RiqiD4mepxfl8N4t>

* Siehe ‚drops‘ Scene
* **Packed\_Scenes**: sind ‚preloads‘ und dienen dem dynamischen Einsatz einer Szene   
  (z.B. ‚collectibles‘ oder auch die Credits\_Scene, die ja erst am Ende des Spiels erscheinen soll

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung Ein Bild, das Text, Screenshot, Multimedia-Software, Software enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

* randomize() sorgt dafür, das bei jedem Start eine andere Zufallszahl generiert wird
* sollen reproduzierbarer Zufallsergebnisse generiert werden (e.g. debugging) kann ein eigener RandomNumberGenerator mit ‚seed‘ verwendet werden
* bei gleichem Seed String wird auch immer die gleiche Zufallszahlenreihe. d.h. im Beispiel die gleiche Positionierung der Drops
* Kontrollierter Zufallszahlengenerator:

*var rnd = RandomNumberGenerator.new()*

*rnd.seed = String ("random\_string").hash()*

*print('First random number ', rnd.randi())  
 print(‘Second random number ', rnd.randi())*

# UI und Control Nodes

Tipp! MSDF einschalten für klare Schrift

Ein Bild, das Text, Software, Multimedia-Software, Multimedia enthält.

Automatisch generierte Beschreibung

**UI – Scene**

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte BeschreibungErgebnis:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Keypoints:

\_ Container wirken sich nur auf (grüne) Kontroll-knoten aus, Icons benötigen TextureRect anstelle eines Sprite2D

\_ Größe und Position wird über Elternknoten gesteuert

# Autoload / Singletons

Im Beispiel „09\_Collectible\_Objects\_v02“ werden Singletons auch zur zentralen Datenhaltung verwendet ...

**Was sind Autoload Singletons?**

* globale Instanz eines Skripts oder einer Szene
* **Autoload**: wird zum Spielstart geladen und ist somit für alle anderen Skripte und Szenen im Projekt verfügbar
* **Singleton:** es wird immer dieselbe Instanz zur Laufzeit verwendet, d.h. Daten oder Zustände im Singleton bleiben über verschiedene Szenen hinweg persistent

**Wie werden Autoload-Singletons eingerichtet?**

1. Scriptewerden keiner Szene zugeordnet, sondern per Autoload geladen
2. Gehe zu ***Projekt > Projekt-Einstellungen > Autoload***
3. Füge ein Skript oder eine Szene hinzu und gib ihr einen globalen Namen   
   (z.B. GameController)
4. Danach können die Inhalte an jeder Stelle im Projekt über diesen globalen Namen aufgerufen werden

**Typische Singletons:**

* *Game Controller* (z.B. Spielerstatus wie Gesundheit, Inventar abspeichern
* *Audio Controller* (z.B. Wechsel der Hintergrundmusik, Lautstärkeregelung, Soundeffekte wie Schritte, Einstellungen im Audio Controller bleiben über Szenen hinweg erhalten)
* *Input Controller* (z.B. Pause, Mini-Map, Erklärungen, Inventarübersicht anzeigen)

Hier im konkreten Beispiel werden nur die Signale in einem Autoload definiert.

# Signale im Spiel

Für Signale gilt immer der Zusammenhang von **drei Schritten:**

# **Definition** im Singleton (z. B. event\_controller.gd)   
signal got\_damaged ()

# Jeder beliebige Knoten kann das Signal **senden**  
EventController.emit\_signal (“got\_damaged")

# Einer oder mehrere andere Knoten (Observer) **verbindet** sich mit dem Signal   
EventController.connect ("got\_damaged", on\_event\_got\_damaged\_speed) ... und   
so kann z.B.. die Geschwindigkeit des Characters und gleichzeitig die Geschwindigkeitsanzeige geändert werden

Signallogik im Überblick (Screenshots werden in eckigen Klammern referenziert)

|  |  |  |  |
| --- | --- | --- | --- |
| **drops.gd** | **player.gd** | **game\_ui.gd** | |
| Generierte Knoten (Ressourcen und Gefahren, näheres im nächsten Abschnitt) werden Gruppen zugeordnet   * Holes * Damage * Boost   Mit diesen Knoten kann kollidiert werden, was dann unterschiedliche Signale triggert. | Der Player Knoten verbindet sich mit den Signalen [1].    Der Sprite2D registriert Kollisionen und triggert entsprechende Reaktionen (func ....) im konkreten Beispiel wird die Geschwindigkeit erhöht oder veringert [2] und das Item verschwindet, nachdem es gesammelt wurde. | | In der Spieleoberfläche werden 3 Parameter angezeigt  Ähnlich zum Player Knoten, verbindet sich der Knoten wieder mit den Signalen.  Hier wird der Spielstand dann entsprechend aktualisiert und angezeigt [3] |

[1] drei Signale werden in player.gd registriert

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

[2] Veränderung der Beschleunigung

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

[3] Spielstandanzeige

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Im zweiten Schritt (‚acceleration‘ hoch und runter setzen) wird die eigentliche Game Mechanik definiert:

* Geschwindigkeit wird über das Einsammeln von Gemüse (wirkt beschleunigend) bzw. einer Kollision mit Bomben und Löchern (wirkt entschleunigend) gesteuert
* Zusätzlich verlieren wir Leben, was schlussendlich zum Spielende führt.
* Die erreichten ‚Veggie Points‘ sind dann der Highscore. Wird oft verwendet um die Wiederspielbarkeit eines Spiels zu verbessern (Spieler:innen kehren immer wieder zurück, um den aktuellen Wert zu übertreffen) 🡺 neue Idee ‚Leaderboard‘ implementieren
* Ausblick: An dieser Stelle setzt dann auch die kritische Reflektion ein, kompetitive Spielelemente sind relativ leicht umzusetzen. Aber welche Anreize (außer einem Leaderboard) könnten zum erneuten Spielen motivieren (z.B. RPGs haben oft mehrere Enden zu entdecken: Das Spiel "Undertale" hat z.Bsp. einen **Pacifist**, **Neutral** oder **Genocide** Durchlauf, ohne diese jedoch explizit einzufordern.
* Undertale (2015): Indie Klassiker, mehrfach Game of the Year, bekannt für besonders innovative Erzählweise (Kämpfe durch Dialoge, Gnade oder geschickte Ausweichmanöver lösen), Musik wird in Live-Konzerten aufgeführt

Ein Bild, das Screenshot, Text, Cartoon, Visitenkarte enthält.

Automatisch generierte Beschreibung

Quelle: <https://www.npr.org/2020/10/17/924581553/make-love-not-war-five-years-of-undertale>

# Audio Play

* Gutes ‚Negativ-Beispiel‘
* Sound Effekte wurden im ‚game\_ui.gd‘ script mit einem zusätzlich ‚AudioStreamPlayer‘ Node eingefügt
  + Geht schneller, ist aber schwerer zu finden, um bei Bedarf das Sound Design anzupassen
  + Refactoring: Längerfristig wäre hier ein ‚SoundController‘ Node sinnvoll, der über Signale angesteuert wird (event. auch als Autoload)
* Zuerst werden alle Soundeffekte geladen (d.h. es wird eine Instanz des MP3 erzeugt) und sind jederzeit im Script verfügbar   
  Andernfalls müssten die Soundeffekte zur Laufzeit geladen werden und dies würden das Spiel entsprechen verzögern ( *var stream = load(audio\_files[sound]* )

Ein Bild, das Text, Schrift, Screenshot enthält.

Automatisch generierte Beschreibung

* Zum Speichern der Soundinstanzen wird ein Dictionary (d.h. ein Key : Value Paar) verwendet. Je nach eintreffendem Signal können dann die Soundeffekte über ihre ‚Keys‘ aufgerufen werden   
    
  Ein Bild, das Text, Screenshot, Schrift enthält.

  Automatisch generierte Beschreibung

# Tweens und die ‚credits\_scene‘

* Tween Erklärung <https://www.youtube.com/watch?v=NB64GQX9mrw>
* Tweens sind leichtgewichtige Animation – Nodes mit vier Parametern:
  + Welches Objekt,
  + soll welche Eigenschaft ändern,
  + mit welcher Zielgröße und
  + wie lang soll diese Animation dauern

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung