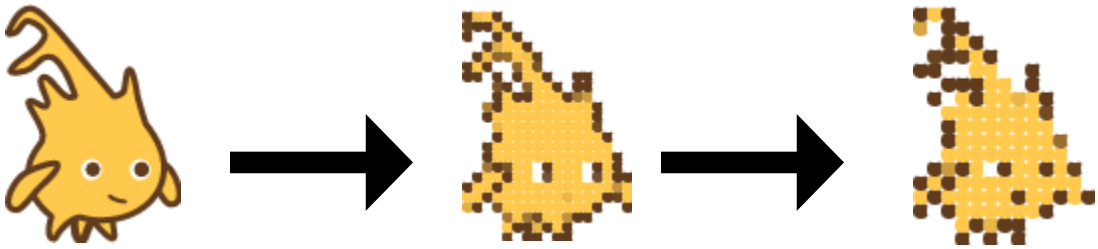
Dieses Beispiel baut auf den Kapiteln
Blockpalette, **Liveness**, und **First-class Objekte** auf!

Pixelize



In diesem Beispiel werden **Pixelgrafiken** thematisiert. Es eignet sich besonders gut als Teil einer Unterrichtseinheit, beispielsweise in Jahrgangsstufe 7 (Dauer: etwa eine Doppelstunde), die Pixel- und Vektorgrafiken aus Jahrgangsstufe 6 aufgreift. Es wird wenig Vorwissen benötigt; dennoch wäre es gut, wenn die Zielgruppe bereits erste Erfahrungen mit blockbasierten Sprachen, z. B. in Form von Scratch gesammelt hat.
Wir verwenden neue Konzepte, die blockbasierte Sprachen uns ermöglichen - wie etwa **Tinkering** und **Direct Drive**. Explorativ werden hier außerdem Konzepte blockbasierter Programmierung wie **Leinwand, Kostüme,** und **First-class Objekte** erforscht.

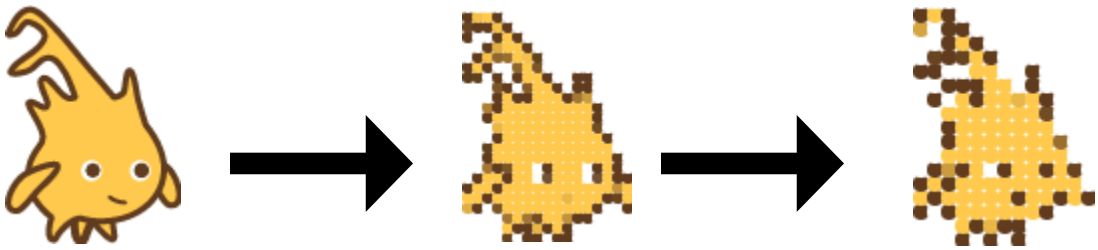
Das Material besteht aus **Schülermaterialien** und **zusätzlichen Anmerkungen für Lehrkräfte** zur Umsetzung im Unterricht sowie **weiterführenden Erklärungen** und **Lösungen.**

Somit ist es möglich, das Material sowohl für die **eigene Weiterbildung** als auch in Auszügen für **Arbeitsblätter für den Schuleinsatz** zu verwenden.

Überblick über die verwendeten Symbole in diesem Handout:

	Aufgabe
	Bonusaufgabe für Fortgeschrittene, bzw. Benutzer mit Vorerfahrung
	Hinweis
	Tinkering: hier gibt es keine falschen Antworten!
	Gespräch: Reflektion, Diskussion oder Austausch mit dem Banknachbarn
	Tipp: Hinweise, bzw. Ideen für die Umsetzung im Unterricht

Pixelize



**In diesem Modul wirst du Pixelgrafiken erzeugen -
und zwar indem du dein eigenes Bild verpixelst!**

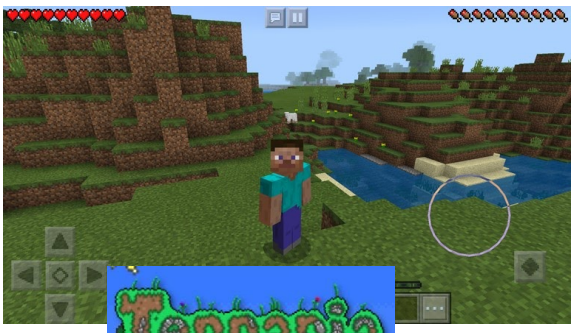
Vorgeschmack auf das fertige Projekt: bit.ly/snap-pixelize-solution

🔗 Link zum Projekt: bit.ly/snap-pixelize

Öffne das Projekt über den Link aus der Box.

Du schlüpfst nun in die Rolle eines **Spieleentwicklers.** Du bist Teil des **Designerteams** und **bist für die Grafik in eurem Spiel zuständig!**

Da euer Budget (noch) begrenzt ist, bietet es sich an, für euer Spiel **Pixelgrafik** zu verwenden - diese Art der Grafik kennen wir beispielsweise aus **Minecraft, Terraria, Stardew Valley, Dead Cells, Shovel Knight, ...**

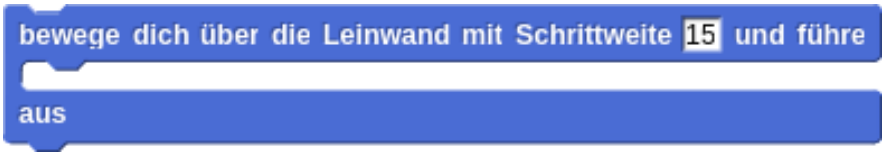


...Und da du clever bist, hast du dich dazu entschlossen, die Pixelgrafiken für euer Spiel in Snap zu entwerfen!

Nachdem wir das Projekt öffnen, sehen wir einige Objekte, sog. **Sprites**. Wir sehen ein Objekt namens **painter**, ein Objekt namens **background** und ein Objekt namens **Stage**, bzw. **Leinwand**.

- Wähle das **painter-Objekt** aus.
- Uns interessiert momentan **nur** dieses “painter”-Objekt - du musst in den anderen Objekten **nichts** tun!
- **Keine Objekte löschen:** Wenn Objekte gelöscht werden, musst du **wieder von vorne anfangen!**

Der **Skriptbereich** (große graue Fläche in der Mitte) zeigt die Blöcke an, die das ausgewählte Objekt “kennt”. Wir führen Blöcke aus, indem wir sie **anklicken**.



Dieser Block wird der Kern dieses Moduls sein!

Aufgabe: Klicke den Block an, um ihn auszuführen.
Notiere, was auf der Leinwand geschieht!

_____

_____

Aufgabe: Der Block verfügt über einen Eingabeparameter, bzw. einen Input. Experimentiere mit verschiedenen Werten.
Notiere, was dieser Parameter bewirkt!

_____

_____

Bonusaufgabe: Genau genommen hat der Block sogar **zwei** Inputs.
Identifiziere den zweiten Input!

_____

Lösungen:

- Das **painter-Objekt** bewegt sich mit einer bestimmten Schrittweite über die Leinwand.
- Der Parameter erhöht die Schrittweite des Objekts. **Nebeneffekt:** eine höhere Schrittweite bedeutet, dass das Objekt weniger Zeit benötigt, um die Leinwand abzulaufen. Daher antworten Schülerinnen und Schüler hier oftmals, dass der Parameter die **Geschwindigkeit** des Objekts erhöht.
- Der zweite Input ist der leere “Mittelteil” des Blocks. Hier kann eine Aktion (= Code) eingefügt werden, die das Objekt bei jedem Schritt durchführt!

Tipps:

Es empfiehlt sich, den Schülerinnen und Schülern etwa 5 Minuten Bearbeitungszeit zu geben.

Abgesehen davon, dass die Schülerinnen und Schüler hier das Verhalten des Objekts beobachten, ist es hier auch wichtig, dass

- sie verstehen, dass **Blöcke per Mausklick direkt ausgeführt werden können - das ist direct drive!**
- sie verstehen, was Inputs und Parameter sind und wie diese in blockbasierten Programmiersprachen dargestellt werden (insbesondere die Bonusaufgabe macht das deutlich: der zweite Input ist nicht einfach ein weißer Slot wie der erste!)

Beim zweiten Input (in der **Bonusaufgabe**) handelt es sich um ein Beispiel für Snap!s *first-class data*. **Der Datentyp des Inputs ist Code.**

>> Weiterführende Informationen und Aufgaben zu First-class Daten: bit.ly/script-fc

Nach Ablauf der Bearbeitungszeit für diese Phase sollten die Ergebnisse **gesammelt und besprochen** werden, um sicherzustellen, dass die gesamte Gruppe mit demselben **Grundwissen weiterarbeitet.**

Tipp: An dieser Stelle bietet es sich im Unterricht an, das in der Informatik grundlegende Prinzip von Eingabe-Verarbeitung-Ausgabe (EVA) zu thematisieren, bzw. aufzugreifen und die einzelnen Aspekte anhand des Blocks zu identifizieren. Verfügt der Block über eine Ausgabe?

Aufgabe: Neben diesem Block benötigen wir einige weitere Blöcke.

Finde die folgenden Blöcke in der Blockpalette und lege sie im Skriptbereich ab:

Du findest diese (und viele andere) Blöcke in der **Blockpalette von Snap!.**

Aufgabe: Klicke auf die Blöcke, um sie auszuführen.

Notiere für jeden Block, was er bewirkt!
Achtung: Du musst die Blöcke NOCH NICHT miteinander kombinieren!

_____

_____

_____

_____

*(Tipp: Bewege das **painter**-Objekt, nachdem du diesen Block geklickt hast!)*

_____

*(Tipp: Benutze zuerst den **stemple**-Block!)*

Tinkering: Kombiniere nun die einzelnen Blöcke zu einem sog. **Skript.**

Führe dein Skript aus. Was geschieht dabei auf der Leinwand?
Schaffst du es mithilfe dieses Skriptes bereits, **die Leinwand mit Stempelabdrücken (= Pixeln) zu füllen?**

Die Leinwand soll außerdem vor jedem Durchgang gewischt werden!

Lösungen:

Objekt zeigt auf den Mauszeiger

Objekt zeigt in eine zufällige Richtung

Objekt zeigt auf die Mitte der Leinwand

Objekt stempelt ein Abbild von sich an der aktuellen Position auf die Leinwand; **oftmals geben hier Schülerinnen und Schüler an, dass das Objekt "kopiert" wird.**
Das ist konzeptionell jedoch falsch!

Gestempelte Abbilder werden entfernt

Mögliche Lösung für die Tinkering-Phase:

Tipps:
Es empfiehlt sich, den Schülerinnen und Schülern etwa 5-10 Minuten Bearbeitungszeit zu geben. Nach Ablauf der Bearbeitungszeit sollten die Ergebnisse **gesammelt und besprochen** werden, um sicherzustellen, dass die gesamte Gruppe mit demselben **Grundwissen weiterarbeitet.**

Es gibt **drei Gründe** dafür, dass Schülerinnen und Schüler die Blöcke **selbst aus der Blockpalette suchen müssen**, anstatt dass sie vorgegeben werden:

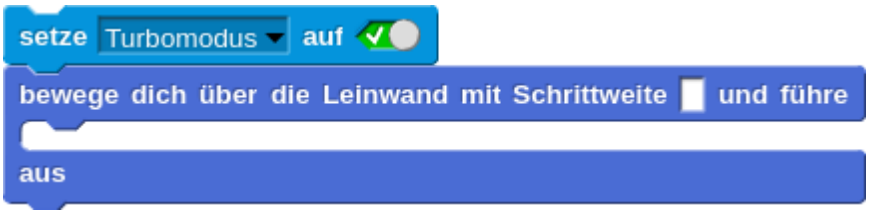
1. Sie müssen sich so mit den verschiedenen Kategorien vertraut machen.
2. Sie stellen dabei fest, dass die Farben der Blöcke zu den Farben der Blockkategorien gehören.
3. Dank diesem Schritt wissen sie, wo sie die Blöcke wiederfinden können, falls sie versehentlich gelöscht werden.

Erneut wird **direct drive** benutzt, d.h. Schülerinnen und Schüler klicken auf die Blöcke um sie auszuführen. Das vermittelt wichtige Vorzüge blockbasierter Programmierung: eine **Direktheit** und **Greifbarkeit** dessen, was auf dem Bildschirm passiert.
Zum Schluss gilt es nun, die einzelnen Blöcke zu einem **Skript** zusammenzubauen. Diese Phase heißt **Tinkering**, da es hier kaum richtige und falsche Lösungen gibt; hier wird durch die Formulierung bewusst kreativer Freiraum gelassen.

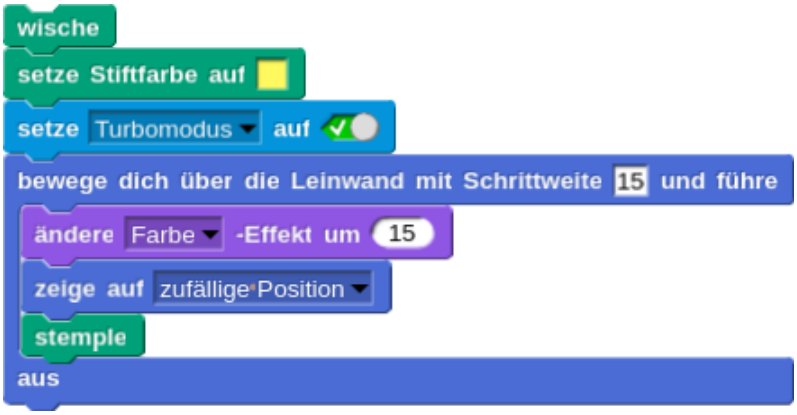
Da wir jetzt wissen, was der Block macht, möchten wir nicht jedes mal zusehen müssen, wie er über die Leinwand wandert.

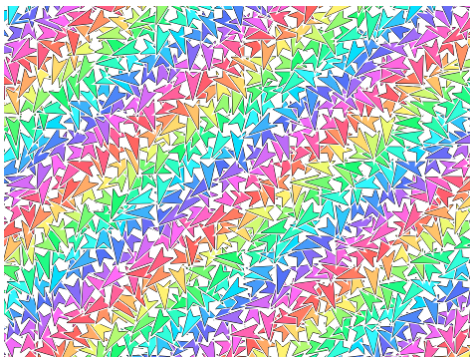
Deshalb verwenden wir den Turbo-Modus (rechts). Wir können den Turbo-Modus einfach mit einem Block aktivieren, den wir in der Kategorie “Fühlen” finden:

!



Mithilfe von Blöcken in den Kategorien **Aussehen** und **Stift** können wir unser Skript um neue Funktionalitäten erweitern. **Beispiel:**





Tinkering: Schaffst du es, das Skript so zu verändern, dass die Pfeile nicht zufällig gedreht sind, sondern alle auf den Mauszeiger zeigen?
Schaffst du es, das Skript so zu verändern, dass die Pfeile **weg** vom Mauszeiger zeigen?

Schaffst du es, das Skript so zu verändern, dass wir es nicht immer anklicken müssen, sondern es z. B. ausgeführt wird, wenn wir die Leertaste drücken?

Tipp: Die Kategorie “Steuerung” der Blockpalette ist ein heißer Tipp!

Bonusaufgabe: Verändere das Skript so, **dass der Farbeffekt verändert wird, wenn das Objekt nahe am Mauszeiger ist,** sodass ein farbiger Kreis um den Mauszeiger herum entsteht!

Tipps:
Wir wenden uns in dieser Phase erneut der **Blockpalette** zu.
Sie gliedert die verfügbaren Blöcke in **semantische Einheiten** und hilft dem Benutzer dabei, Blöcke zu finden, **selbst wenn man nicht weiß, wonach man sucht.**

>> Weiterführende Informationen und Aufgaben zur Blockpalette: bit.ly/script-pal

In der Tinkering-Phase wird das Skript interaktiver gemacht. Dazu verwenden wir Ereignisse.
An dieser Stelle bietet es sich daher im Unterricht an, die ereignisgesteuerte Programmierung in Snap! mit der **main**-Methode-Programmierung z. B. in Java zu vergleichen. Skripte werden ausgeführt, wenn Ereignisse eintreffen. Werden Skripte, bzw. Blöcke einfach angeklickt, um sie auszuführen, sprechen wir von **Direct Drive** - einem Spezialfall der ereignisgesteuerten Programmierung. Direct Drive basiert auf der **direkten Manipulation** von Programmen und ihrem Code und findet sich in vielen Programmiersprachen der blockbasierten Scratch-Sprachenfamilie.

>> Weiterführende Informationen und Aufgaben zum ereignisgesteuerten Programmierparadigma: bit.ly/script-event

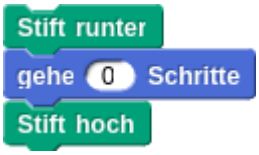
Die Bonusaufgabe erfordert eine **bedingte Anweisung** und ist daher für besonders **schnelle** Schülerinnen und Schüler, und solche mit **Vorerfahrung** intendiert.

Mögliche Lösung nach Tinkering und Bonusaufgabe:

Bis hierhin haben wir das “painter”-Objekt als **Stift** verwendet. Wir **haben dabei quasi “Pixel” auf die Leinwand gestanzt, um Bilder zu erzeugen.**

Im fertigen Spiel wollen wir aber keine Dreiecke, sondern Pixel haben. Können wir mit dieser Technik auch “richtige” Pixelgrafiken erstellen?

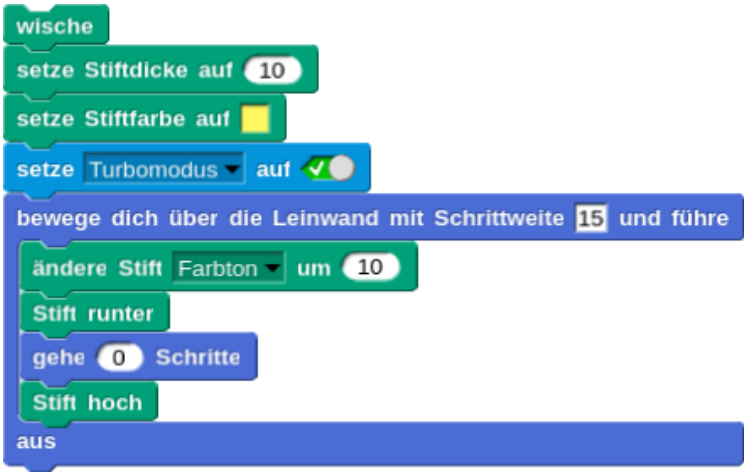
Um das herauszufinden, experimentieren wir mit einigen Blöcken der Stift-Kategorie:



Hiermit können wir Punkte auf der Leinwand malen!

Wir müssen also unseren bisherigen Code abändern, bzw. teilweise löschen.

Wir können beispielsweise das folgende Skript erzeugen.



Aufgabe:
Baue diesen Code nach und notiere, was er bewirkt!

_____

_____

_____

_____

_____

Aufgabe: Notiere, was der “ändere Stift (Farbton)”-Block im oberen Skript bewirkt!

_____

_____

Notiere die weiteren Eigenschaften, die wir mithilfe des Blocks ändern können!

_____

Notiere, was die Eingabeparameter Stiftdicke und Schrittweite am fertigen Bild verändern!

_____

_____

Tipps:

Die Konstruktion Stift runter-gehe 0 Schritte-Stift hoch ist ein notwendiger “Workaround”, damit ein farbiger Punkt an der aktuellen Position gemalt wird, ohne dass das Objekt sich dabei bewegt.
Die Konstruktion ist aus dem Kapitel zu **Liveness** bekannt.

>> Weiterführende Informationen und Aufgaben zur Liveness: bit.ly/script-liveness

Hier wird zunächst das Problem identifiziert - wir wollen im fertigen Bild keine Dreiecke, sondern Pixel. Daher muss der Code angepasst werden.

Das Problem **aufzuzeigen** und dann **nach einer Lösung zu suchen**, zeigt den Schülerinnen und Schülern, dass eine “Lösung” schrittweise entwickelt werden kann, und dass es nicht schlimm ist, in eine Lage zu geraten, wo große Änderungen vorgenommen werden müssen - wir demonstrieren und ermutigen hier und anhand der gestellten Aufgaben eine **explorative, spielerische Herangehensweise an die Programmierung.**
Ein noch deutlicheres Beispiel hierfür folgt im nächsten Abschnitt.

Lösungen:

- Es können diverse Eigenschaften des Stifts (und somit der einzelnen Punkte) verändert werden, was das finale Bild auf der Leinwand verändert
- Die Bestandteile des HSB-Farbraums, also Farbton, Sättigung, Helligkeit *(engl. HSB = hue, saturation, brightness)* und Transparenz
- Stiftdicke verändert die Größe der einzelnen Punkte, Schrittweite verändert den Abstand der Punkte zueinander. Indirekt verändern beide Parameter außerdem die Größe der weißen Abstände zwischen Punkten, und die Malgeschwindigkeit

Damit können wir malen.

...leider sind wir damit noch meilenweit davon entfernt, gute Pixelgrafiken für Spiele entwerfen zu können. **Offensichtlich kommen wir mit unserem farbigen Punkten momentan nicht weiter!**

Wir brauchen einen Plan B.
Können wir existierende Bilder vielleicht einfach abpausen?

Schauen wir uns das Objekt "background" und seine Kostüme an. Hier finden wir einige Bilder, die wir einfach **abpausen** können!

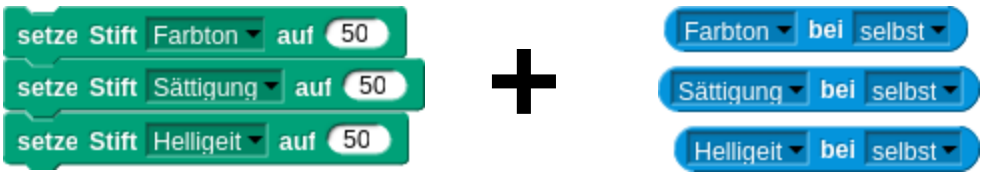
! Im background-Objekt muss nichts getan werden! Diese Hintergrundbilder dienen nur als Blaupause, bzw. Vorlage und müssen nicht auf die weiße Leinwand gezogen werden.

Wie gehen wir genau vor wenn wir abpausen? Wir schauen uns eine Stelle an, merken uns, welche Farbe wir dort im Original vorfinden, und kopieren diese dann auf unseren Entwurf.

Wie übertragen wir diesen Vorgang in Snap?

- Wir bewegen uns schrittweise über das jeweilige Bild
- Wir merken uns, welche Farbe der Pixel im Original hat
- Wir malen einen Pixel mit genau dieser Farbe auf unseren Entwurf

Snap! verwendet für die Definition, bzw. Kodierung von Farben das HSB-Modell (Hue-Saturation-Brightness; **Farbton, Sättigung** und **Helligkeit**):



Aufgabe: Füge die einzelnen Elemente nun zu einem Skript zusammen, das

- die Leinwand sauber wischt
- die Größe des Pinsels setzt (z. B. auf 15)
- schrittweise über die Leinwand läuft (Schrittweite z. B. 15)
- sich die jeweiligen Farbwerte an der aktuellen Position merkt und die Pinselfarbe entsprechend setzt
- einen Farbpunkt an der aktuellen Position malt

Bonusaufgabe: Der Benutzer soll in der Lage sein, **per Druck auf die Pfeiltasten die Größe der Pixel zu verändern.**

Implementiere dazu Skripte, die auf Pfeiltastendruck reagieren, und die "Pixelgröße" (= Schrittweite) verändern.

Tipps:

Erfahrungsgemäß haben manche Schülerinnen und Schüler hier ein Verständnisproblem mit dem Wort “abpausen”. Daher empfiehlt es sich, kurz im Plenum zu besprechen, was damit gemeint ist.

Der Großteil der Schülerinnen und Schüler ist dann in der Lage, die Aufgabe zu lösen, insbesondere nachdem das Skript in der vorherigen Aufgabe nachgebaut und seine Bestandteile erfasst wurden.

Lösungen:

- Um die Aufgabe zu lösen, müssen lediglich die angegebenen Blöcke miteinander kombiniert und an die Stelle des “ändere Stift-Farbton um”-Block gesetzt werden
- Oft verwenden Schülerinnen und Schüler fälschlicherweise den 

-Block anstelle von 

- Zur Lösung der Bonusaufgabe müssen Skripte implementiert werden, die eine Variable (z. B. “Pixelgröße”) erhöhen, bzw. verringern, und dann das “Mal-Skript” aufrufen (in dem wiederrum Schrittweite und Stiftgröße von der Variable abhängen).

Das Skript verfügt über **zwei** besonders wichtige Parameter: **Stiftgröße** und **Schrittweite**.

Tinkering: Experimentiere mit unterschiedlichen Werten für diese beiden Parameter.
Was kannst du bei besonders niedrigen Werten beobachten?
Was bei besonders hohen Werten?
Müssen beide Parameter stets denselben Wert haben?
Wie wirken sich Veränderungen dieser Parameter auf die Ausführgeschwindigkeit und die Qualität des fertigen Bildes aus?

Abschließendes Tinkering:

Ziehe jetzt per **Drag-and-Drop** deine eigenen Bilder (z. B. Google-Bildersuche!) in das **Kostüm-Menü des background-Objektes!**

Gib dem Objekt das richtige Kostüm und **überprüfe, ob dein Programm das Bild korrekt verpixelt!**

Schaffst du es, deinen (verpixelten) Bildern Grafikeffekte und Farbeffekte zu geben?

Reflektion

Was haben wir in diesem Abschnitt konzeptionell getan?
Wir haben eine Pixelgrafik erzeugt, indem wir uns schrittweise über eine Leinwand bewegt und Farbpunkte gesetzt haben. Die Schrittweite bestimmt dabei die “Auflösung”, d.h. Anzahl der Pixel.

Indem wir Schrittweite und Größe der Farbpunkte (Pinselgröße) verändern, verändern wir somit die “Qualität” unseres Bildes. Eine hohe Qualität, d.h. kleine Schrittweiten und viele winzige Farbpunkte bewirken, dass die Erzeugung der Grafik länger dauert. Gleichzeitig bedeutet es auch, dass ein Computer beim Speichern dieser Grafik viel mehr Informationen verarbeiten muss, als bei einer größeren Schrittweite und Pinselgröße.

Abschließende Frage/Reflektion/Hausaufgabe zum Überlegen:
Könnten wir unser Bild jetzt strecken, d.h. “größer machen”?

Tipp:
Die erste Tinkering-Phase leitet Schülerinnen und Schüler dazu an, die Prinzipien ihres Programms zu reflektieren. Die Ergebnisse dieser Phase eignen sich daher dazu, **im Plenum besprochen zu werden.**

Lösung:
Um Bilder nicht nur zu verpixeln, sondern ihnen “Grafikeffekte” zu verleihen, müssen lediglich kleine Änderungen vorgenommen werden, wie z. B. Multiplikation/Addition/Subtraktion des Farb-, Sättigungs- oder Helligkeitswertes im Originalbild:

Tipp:
Am Ende bietet es sich im Unterricht an, aufzugreifen, wie Vektorgrafiken erzeugt, bzw. beschrieben werden.
Inwiefern unterscheiden sie sich von unseren Pixelgrafiken?