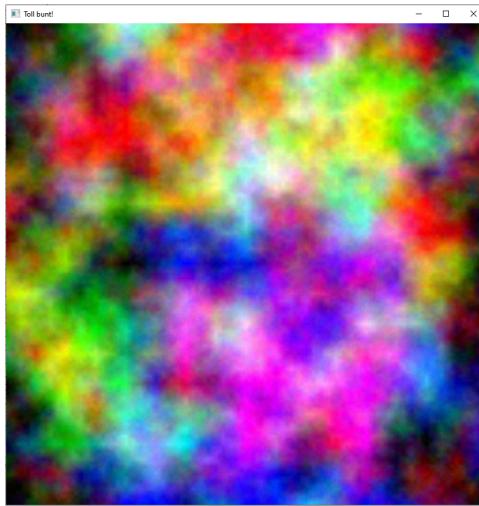


Übungsblatt 5

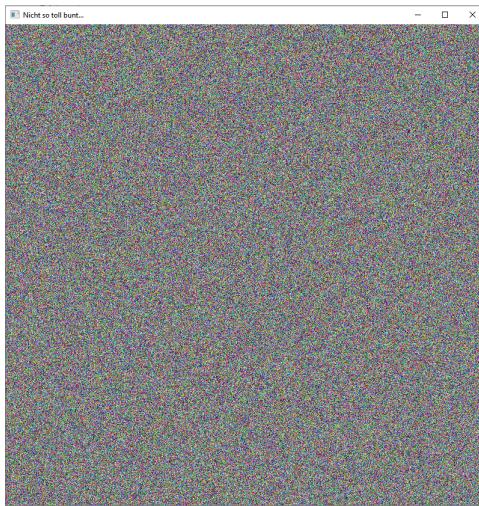
Da die letzte Vorlesung wegen des Hochschultags ausgefallen ist, gibt es auf diesem Übungsblatt nur Bonusaufgaben. Ich empfehle aber jeder und jedem es trotzdem zu bearbeiten, insbesondere dann, wenn Sie Schwierigkeiten mit der Programmierung haben!

Bei dieser Übung geht es darum, ein buntes Bild zu erzeugen, denn wie Konfuzius Nichte schon gut hätte sagen können: „Bunt ist immer gut!“ Sie sollen in dieser Übung ein Programm schreiben, das Bilder ähnlich dem Folgenden generiert:



Wir sehen uns zunächst an, wie die prinzipielle Vorgehensweise bei der Erstellung eines solchen Bildes ist. Dann folgen konkrete Aufgaben zur Umsetzung dieser Vorgehensweise.

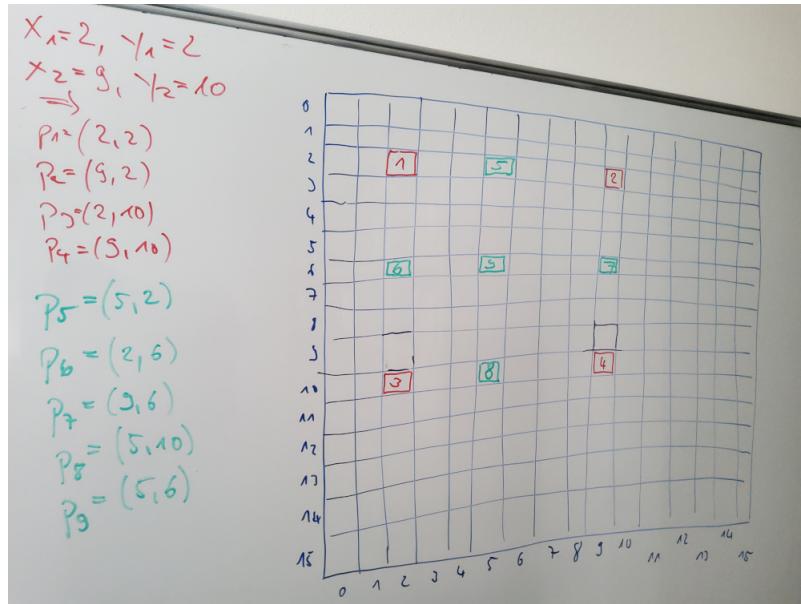
Auf dem Bild sind zufällig Farben ineinander verwürfelt und verschmiert. Das gelingt aber nicht, indem man die Pixel nur nach Zufallsprinzip bunt färbt, denn dann ergäbe sich ein Effekt wie der Folgende:



Das ist zwar auch bunt, aber das Bild zeigt einfach nur Farbrauschen und macht nicht viel her.

Im ersten Bild werden zwar auch Punkte zufällig gesetzt, es wird aber darauf geachtet, dass dass die farbliche Abweichung von Pixeln, die nahe beieinander liegen gering ist. Wenn Pixel hingegen weit auseinander liegen, können sie auch sehr unterschiedliche Farben annehmen.

Die Vorgehensweise zur Generierung ist dabei wie folgt:



Prinzipielle Vorgehensweise; durch das blaue Raster seien die Pixel eines Bildes dargestellt:

1. Ausgangslage ist ein rechteckiger Ausschnitt des Bildes und in diesem die Farbe der Punkte, die am weitesten auseinander liegen: Das sind die vier Eckpunkte des Bildausschnitts (in der obigen Grafik sind diese vier roten markierten Pixel mit den Nummern 1, 2, 3, 4).
2. Danach werden die grün markierten Pixel gemalt (5, 6, 7, 8, 9) und zwar wie folgt: Um Punkt 5 zu erzeugen, werden die Farbwerte der Pixel 1 und 2 zunächst gemittelt und der resultierende Mittelwert wird anschließend zufällig verfälscht. Die Größe der Verfälschung hängt davon ab, wie weit die Pixel 1 und 2 auseinander liegen: Liegen sie weit auseinander, darf die Verfälschung groß sein, für sehr nahe zusammen liegende Pixel darf die Farbe kaum verfälscht werden.

Auf die gleiche Weise entsteht Pixel 6 aus Pixel 1 und 3, der Pixel 7 entsteht aus 2 und 4 und der Pixel 8 entsteht aus 3 und 4.

((*) Den folgenden Satz verstehen Sie vielleicht erst, nachdem Sie den 3. Schritt gelesen haben: Alle diese Punkte dürfen nur erzeugt werden, wenn sie noch nicht anderweitig vorher erzeugt wurden.)

Der Pixel 9 wird ähnlich gebildet, seine Ausgangsfarbe vor der Verfälschung wird allerdings als Mittelwert aus allen 4 Pixeln 1, 2, 3, 4 errechnet.

3. Dadurch wird der Ausschnitt in vier Unterausschnitte aufgeteilt, nämlich

- den mit den Eckpunkten 1, 5, 6, 9 (erster Unterausschnitt),
- den mit den Eckpunkten 5, 2, 9, 7 (zweiter Unterausschnitt),
- den mit den Eckpunkten 6, 9, 3, 8 (dritter Unterausschnitt) und
- und den mit den Eckpunkten 9, 7, 8, 4 (vierter Unterausschnitt).

Für jeden dieser Unterausschnitte sind mindestens die Farben der Pixel in den Ecken nun vorgegeben und für jeden dieser Unterausschnitte wird das Verfahren rekursiv wiederholt. Der Rekursionsabbruch erfolgt, wenn der Abstand der Eckpunkte (in x - und in y -Richtung) gleich 0 ist.

Dabei ist zu beachten, dass sich die Unterausschnitte jeweils leicht überlappen. Zum Beispiel teilt sich der Unterausschnitt 1 mit dem Unterausschnitt 2 die Pixelspalte zwischen den Pixeln 5 und 9. Das ist wichtig, damit die Farben zwischen den beiden Ausschnitten sanft ineinander verlaufen können.

Das führt auch dazu, dass zum Zeitpunkt, zu dem etwa Unterausschnitt 2 gemalt werden soll, diese Pixelspalte (zwischen den Pixeln 5 und 9) bereits komplett ausgefüllt sein kann (das könnte bei Generierung des Unterausschnitts 1 passiert sein), und nicht übermalt werden darf. (Jetzt können Sie noch einmal den Satz aus Schritt 2 lesen, der mit einem (*) markiert ist.)

Aufgaben:

1.

(Bonusaufgaben)

Zunächst erweitern wir die Klasse `Color`, die wir auf Übungsblatt 4 erstellt haben.

- (a) Fügen Sie dieser Klasse eine Methode mit dem Kopf

```
Color averageWith(const Color& other) const
```

hinzu, die den Mittelwert der Farbe `this` und `other` ermittelt, indem die drei Farbkomponenten (Rot, Grün, Blau) jeweils gemittelt werden.

- (b) Fügen Sie, analog dazu eine weitere Methode mit dem Kopf

```
Color averageWith(const Color& other1, const Color& other2, const Color& other3) const
```

hinzu, die den Mittelwert der vier Farben `this`, `other1`, `other2` und `other3` ermittelt.

- (c) Fügen Sie eine weitere Methode mit dem Kopf

```
Color randomlyChange(int maxAmount) const
```

hinzu, die aus der Farbe `this` eine neue Farbe erzeugt, bei der alle Farbkomponenten zufällig verfälscht werden (zum Beispiel wird der rote Farbanteil erhöht oder erniedrigt). Die Verfälschung in jedem Farbanteil soll aber nicht größer sein als `maxAmount`. Außerdem dürfen auch die so verfälschten Farbanteile das Intervall $[0, 255]$ nicht verlassen.

Hinweis: Zur Erzeugung von Zufallswerten können Sie die Funktion `rand()` aus der Bibliothek `algorithm` benutzen. Diese Funktion liefert eine „Zufallszahl“ aus dem Bereich $[0, 2^{15} - 1]$ zurück, also eine Zufallszahl mit 15 zufällig ausgewählten Bits.

Die von `rand()` erzeugten „Zufallszahlen“ sind nicht wirklich zufällig, da sie von einem sogenannten Pseudozufallszahlengenerator erzeugt werden. So ein Generator liefert nicht wirklich Zufallszahlen sondern nur eine Abfolge von Zahlen, die auf den ersten Blick zufällig wirken. Insbesondere ist diese Folge sogar deterministisch, d.h. es werden pro Programmstart immer die selbe Folge von „Zufallszahlen“ erzeugt. Um jeweils pro Programmstart eine neue Folge von Zahlen zu erhalten, muss der Zufallszahlengenerator jeweils mit einem anderen Wert initialisiert werden (die sogenannte *Saat* (engl. *seed*)). Sie erreichen das z.B. durch den folgenden Befehl:

```
srand(static_cast<unsigned int>(time(nullptr)));
```

Dieser greift die aktuelle Systemzeit ab und benutzt diese als Saat für den Pseudozufallsgenerator. Er muss nur einmal pro Programmstart ausgeführt werden!

- (d) Fügen Sie der Klasse eine Methode mit dem Kopf

```
inline bool equals(const Color& other) const
```

hinzu, die genau dann `true` zurückliefert, wenn die Instanz `this` mit der Instanz `other` in allen drei Farbkomponenten übereinstimmt.

- (e) Fügen Sie der Klasse ein öffentliches, statisches, konstantes Attribut `nullColor` vom Typ `Color` hinzu. Definieren Sie dieses Attribut so, dass es sich von allen vternünftigen Farben unterscheidet, etwa indem Sie alle seine Farbkomponenten auf -1 setzen. Diese Konstante habe die Semantik eines noch nicht initialisierten Farbwerts; wir benötigen sie später.

2.

(Bonusaufgaben)

Ein wesentliches Problem bei der Durchführung des oben beschriebenen Algorithmus zur Erzeugung des bunten Bildes ist, dass Pixel nicht nur geschrieben, sondern auch gelesen werden müssen. Die zur Verfügung gestellte Klasse `ViewPortGL` bietet dafür aber keine Methoden. Das heißt, dass wir eine Klasse brauchen, in der wir uns merken, wie die Farbwerte der einzelnen Pixel gerade sind. Diese Klasse nennen wir `ColorBuf`. Sie finden im Moodle die Headerdatei für diese Klasse.

- (a) Implementieren Sie Konstruktor und Destruktor für diese Klasse! Im Konstruktor

```
ColorBuf(unsigned int widthP, unsigned int heightP)
```

muss dabei ein zweidimensionales Array erzeugt und dem Attribut `colorState` zugewiesen werden. Die Arraygröße in der ersten Dimension ist dabei gleich `widthP` die Arraygröße jeden Arrays der zweiten Dimension ist gleich `heightP`. Der Konstruktor muss zudem die Attribute `width` und `height` richtig initialisieren.

In dem Attribut `colorState` sollen die eigentlich Farbinformationen für Pixel gehalten werden.

Der Destruktor muss wie üblich dafür sorgen, dass alle im Konstruktor reservierten Ressourcen wieder frei gegeben werden.

- (b) Implementieren Sie die getter- und setter-Methoden, die im Header deklariert sind!
- (c) Implementieren Sie die Methode `clear`, die jeden Farbwert des `ColorBuf` auf `Color::nullColor` setzt.
- (d) Implementieren Sie die Methode

```
void drawTo(ViewPortGL& vp)
```

die alle in dem `ColorBuf` gespeicherten Farbwerte als Pixel in den gegebenen `ViewPortGL` schreibt. Die Methode soll einen aussagekräftigen `logic_error` werfen, wenn `ColorBuf` und `vp` unterschiedliche Größen haben.

Sollte einer der Pixel im `ColorBuf` den Farbwert `Color::nullColor` besitzen, so soll die Methode nicht abbrechen, sondern diesen Farbwert ignorieren!

Achten Sie darauf, dass im `ViewPortGL` nur eine endliche Anzahl von Pixeln mit `preparePixel` zwischengespeichert werden können, bevor sie mit `sendPixels` an die Grafikkarte geschickt werden müssen. Sie können diese Anzahl über die Methode

```
ViewPortGL::getMaxNumberOfPreparedPixels()
```

erfragen. Schicken Sie nicht jeden Pixel einzeln ab, das ist höchst ineffizient!

3.

(Bonusaufgaben)

Nach diesen Vorbereitungen sind wir nun dazu in der Lage das bunte Bild zu erzeugen. Gehen Sie dabei wie folgt vor: Erzeugen Sie eine Instanz von `ViewPortGL` und eine von `ColorBuf` (mit gleichen Abmessungen!). Der Aufbau des Bildes geschieht vollständig im `ColorBuf`; erst wenn das Bild komplett aufgebaut ist, wird es per `ColorBuf::drawTo` an den `ViewPortGL` geschickt.

Bevor Sie die utige Methode aufrufen, sollten Sie zuerst den `ColorBuf` mit `clear` aufräumen und alle vier Eckpunkte mit völlig zufälligen Farben befüllen.

- (a) Schreiben Sie eine Funktion mit dem Kopf

```
void buildColorPic(unsigned int x1, unsigned int y1, unsigned int x2,  
                   unsigned int y2, ColorBuf& cBuf)
```

die das Bild in `cBuf` rekursiv aufbaut. Zur Erläuterung der Vorgehensweise beziehen wir uns im Folgenden wieder auf die Skizze oben auf Seite 2 dieser Übung.

Die Parameter `x1`, `y1`, `x2`, `y2` geben den aktuellen Ausschnitt an, der befüllt werden soll.

Die Funktion kann davon ausgehen, dass in den Eckpunkten des spezifizierten Ausschnittes bereits gültige Farben gesetzt sind. Diese Farben können über die Methode `ColorBuf::get` ausgelesen werden.

Sie bestimmt weiterhin die Koordinaten der 5 „grünen Punkte“ (Skizze!). Für jeden Koordinaten wird zunächst geprüft, ob an der entsprechenden Stelle schon ein gültiger Farbwert sitzt (Vergleich mit `nullColor`). Falls nicht, so wird ein entsprechender neuer Farbwert bestimmt (Farben mitteln, Mittelwert zufällig abweichen lassen) und in den `ColorBuf` eingesetzt (mit `ColorBuf::set`).

Danach sind wir sicher, dass sowohl die „roten“ als auch die „grünen“ Punkte einen gültigen Farbwert besitzen und wir können die Methode rekursiv für die Unterausschnitte aufrufen.

Die Rekursion bricht ab, wenn der durch `x1`, `y1`, `x2`, `y2` definierte Ausschnitt sowohl in `x`- als auch in `y`-Richtung keinen Platz für neue Pixel mehr bietet.

Achtung: Es kann passieren, dass z.B. `x1` unmittelbar neben `x2` liegt, (etwa wenn `x1 = 5` und `x2 = 6`), aber `y1` nicht unmittelbar über `y2` (etwa wenn `y1 = 2` und `y2 = 5`). In so einem Fall müssen nicht 5 „grüne“ Pixel neu gesetzt werden, sondern nur 3!

Testen Sie Ihren Algorithmus, indem Sie das Bild aus dem `ColorBuf` an den `ViewPortGL` schicken!

- (b) Falls Sie Übungsblatt 4 gelöst haben, verfügen Sie über die Klasse `GridViewer`. Man kann ein buntes Bild auch über die Zellen des `GridViewer` anzeigen lassen. In dem Fall brauchen wir auch den Zwischenpuffer `ColorBuf` nicht, weil die Farbwerte direkt über den `GridViewer` gelesen und geschrieben werden können. Erzeugen Sie ein buntes Bild nach dem obigen Algorithmus ohne `ColorBuf` in einem `GridViewer`!

