

Visuelle Programmierung

Leon Thomm

März 2019

Inhaltsverzeichnis

1 Kurzfassung	3
2 Grundlagen	4
2.1 Zur Datenverarbeitung im menschlichen Gehirn	4
2.2 Beanspruchung des menschlichen Gehirns bei konventioneller textbasierter Programmierung	5
2.2.1 Komplexität und Informationsinhalt von Zeichen	5
2.3 Effizientere und nachhaltigere Gestaltung des Programmierprozesses durch Graphik	5
2.3.1 Verknüpfung von Bildern zu Inhalt	6
2.3.2 Spaß beim Programmieren	6
2.3.3 Rahmenbedingungen für visuelle Programmiersprache	7
2.4 Mögliche Funktionsweisen	7
2.4.1 Node-Based Visual Programming	7
2.4.2 Sonstige	8
2.4.3 Bekannte Beispiele	8
2.4.4 Fazit	9
3 Umsetzung	10
3.1 Motivation	10
3.2 Node-Based Visual Programming im Detail	10
3.3 Tools	11
3.4 Überblick	12
3.5 Funktionsweise des Key-Features Codegenerierung	13
3.5.1 Beispiel für normale Anweisungen	14
3.5.2 Vollständiges Beispiel	15
3.6 Skalierbarkeit	16
4 Fazit und Ausblick	18

1 Kurzfassung

Inspiriert durch offensichtliche, menschlich-biologische Vorteile der graphischen Strukturerkennung basiert die visuelle Programmierung auf der Idee, Programmcode in einer Art und Weise darzustellen, mit der das Gehirn des Programmierers besser umgehen kann, als mit reinem Text. Im Rahmen meines Projektes habe ich - aufgrund vergeblicher Suche nach einem solchen System - ein Programm programmiert, in welchem es möglich ist indirekt in jeder textuellen Programmiersprache visuell (in erster Linie imperativ) zu programmieren und automatisch den Quellcode generieren zu lassen, orientiert an einem bereits bekannten visuellen Programmiersystem. Ich habe Rahmenbedingungen bezüglich der nötigen Eigenschaften einer solchen Software definiert und mich etwas tiefer in entsprechende Bibliotheken und Sprachen (hauptsächlich Qt und C++) eingearbeitet. Mit dieser Software habe ich also die prinzipielle, realistische Umsetzbarkeit einer limitierungsfreien Programmierumgebung für visuelle Programmierung bewiesen.

2 Grundlagen

Zu Anfang möchte ich kurz erwähnen, dass der Leser durch diese Arbeit nicht das tatsächliche Potential der Technik erfahren kann, welches die Grundlage für die Sinnhaftigkeit ist. Genauso, wie niemand Quelltext verstanden hat, nachdem er zum ersten mal welchen gesehen hat, braucht es auch bei der visuellen Programmierung einiges an Erfahrung um das Paradigma zu verinnerlichen und die gute Funktionalität erfassen zu können.

2.1 Zur Datenverarbeitung im menschlichen Gehirn

Das menschliche Gehirn hat Stärken und Schwächen. Dass es konventionelle Aufgaben von Computern nicht gut „lösen“ kann und viel länger braucht erscheint offensichtlich unter dem Aspekt, dass Computer für genau die Aufgaben geschaffen wurden, die ein Mensch nicht so gut kann, sonst wären sie nicht so nützlich. Eine Sache, die das Gehirn sehr gut kann ist mit Bildern bzw. Graphiken umzugehen. [BEWEIS] Werfen Sie einen Blick auf das folgende Bild:

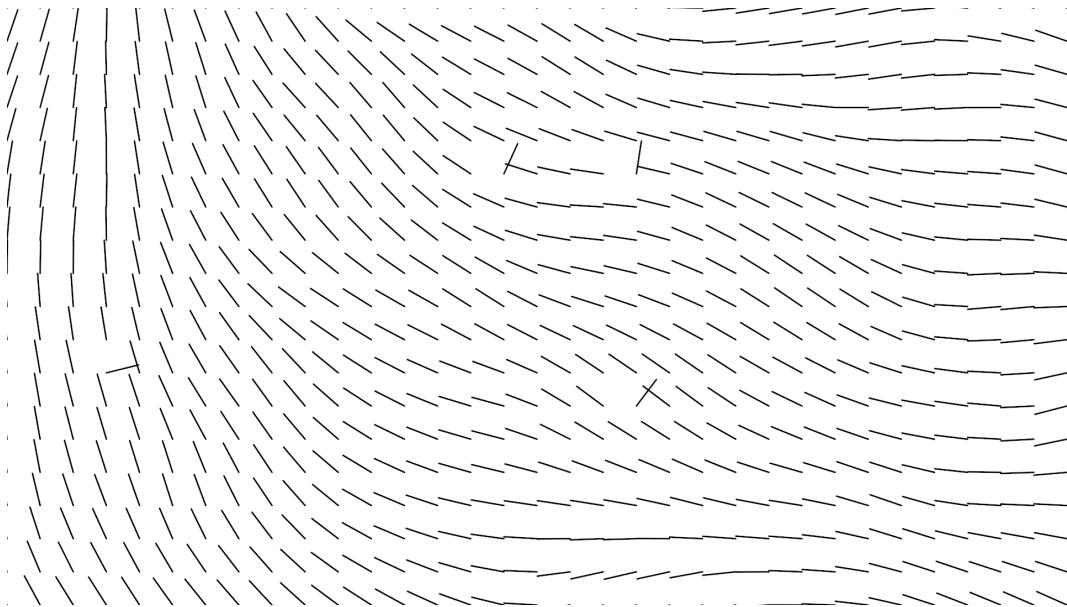


Abbildung 2.1: Beispielbild Strukturerkennung

Ich vermute, Sie haben sehr schnell festgestellt, dass es in diesem Bild Striche gibt, die entgegen ihrer Feldrichtung stehen. Dieses Bild enthält etwa 570 Striche, aus denen Ihr Gehirn gerade genau 4 besonders charakteristische identifiziert hat und das in einer sehr kleinen Zeitspanne.

Nun das Gegenbeispiel: Die nebenstehende Graphik zeigt aufsteigende Zahlen. Können Sie auch hier instinkтив erkennen, welche Elemente nicht rein passen?

```
0001 0002 0003 0004 0005 0006 0007 0008 0009 0010 0011 0012  
0013 0014 0015 0016 0017 0018 0019 0020 0021 0022 0023 0024  
0025 0026 0027 0028 0029 0030 0031 0032 0033 0034 0111 0036  
0037 0038 0039 0040 0041 0042 0043 0044 0045 0239 0047 0048  
0049 0050 0051 0052 0053 0054 0055 0056 0057 0058 0059 0060  
0061 0062 0063 0064 0065 0066 0067 0068 0069 0070 0071 0072  
0073 0074 0075 0076 0077 0078 0079 0080 0081 0082 0083 0084  
0085 0086 0087 0088 0089 0090 0091 0092 0093 0094 0095 0096  
0097 0098 0099 0100 0101 0102 0103 0104 0105 0106 0107 0108  
0109 0110 0111 0112 0113 0114 0115 0116 0117 0118 0119 0120  
0121 0122 0123 0124 0125 0126 0127 0128 0129 0130 0131 0132  
0133 0134 0135 0136 0137 0138 0139 0140 0141 0142 0143 0144  
0145 0196 0147 0148 0149 0150 0151 0152 0153 0275 0155 0156  
0157 0158 0159 0160 0161 0162 0163 0164 0165 0166 0167 0168  
0169 0170 0171 0172 0173 0174 0175 0176 0177 0178 0179 0180
```

Abbildung 2.2: Beispielbild Strukturerkennung Zahlenfolge

Vermutlich nicht, da kein einheitliches graphisches Muster erkennbar ist.

2.2 Beanspruchung des menschlichen Gehirns bei konventioneller textbasierter Programmierung

Bei konventioneller Programmierung wird für Programme eine Darstellungsart benutzt, die der Computer einfach algorithmisch übersetzen kann und die der Programmierer weitgehend problemlos lesen kann. Ganz normaler Text, also das Informationsmedium, mit dem Sie sich gerade befassen. Für das Gehirn kann Text als Informationsrepräsentationsmedium jedoch Nachteile haben.

Das Problem ist auf eine zu schnell zu hohe Homogenität des Aussehens eines Programmscripts reduzierbar, weshalb das Gehirn seine Fähigkeiten der individuellen Bilderkennung kaum nutzen kann.

2.2.1 Komplexität und Informationsinhalt von Zeichen

Text besteht aus aneinandergereihten Zeichen, welche die Eigenschaft besitzen neben eines stark standardisierten und somit nicht individuellen Aussehens (gleiche Größe, Farbe, Dicke etc.), welches Fluch und Segen zugleich ist, auch ein schlechtes Verhältnis zwischen Informationen und graphischer Komplexität zu haben. Die Form eines Zeichens wie z.B. die der Zahl 7 ist bereits deutlich zu komplex angesichts der kleinen Information, die die Zahl repräsentiert. Dadurch ist das graphische Bild des Quellcodes, welches entsteht durch verfassen der benötigten Informationen in Text, bereits zu schnell zu komplex und gleichzeitig zu homogen.

Außerdem hat das (intuitive) Aussehen der dargestellten Informationen für den Programmierer keinerlei Bezug zu diesen selbst. Um sich als Programmierer mit bereits bestehendem Programmcode zu befassen um ihn zu verändern oder zu erweitern muss er die Informationen verstehen, was bei Text deshalb Abstrahieren der eigentlichen Informationen aus dem Text, also aktives lesen bedeutet, da bereits eine kleine graphische Änderung einen unverhältnismäßig starken Einfluss auf den Inhalt haben kann ('0' != 'O').

2.3 Effizientere und nachhaltigere Gestaltung des Programmierprozesses durch Graphik

Es ist immer eine gesunde Balance zwischen Individualität des Aussehens des Programmscripts und Einheitlichkeit (bzw. Standardisierung) zu finden. Diese Balance ist bei Text nicht gut. Durch die zu schnell zu hohe Homogenität des Codes bei textuellem Programmcode, muss dieser immer wieder nach einzelnen Aktionen (z.B. die Änderung des Wertes einer Variable) durchsucht und dafür immer wieder aufwendig gelesen werden. Und da das Bild des dem Programmierer vorliegenden Scripts alles ist, was dieser davon wahrnehmen kann, ist er auch kaum in der Lage verschiedene Stellen in ihrer Bedeutung anhand des intuitiven Aussehens des Scripts (also ohne lesen) zu differenzieren, wenn diese graphische Differenzierung aufgrund genannter Nachteile nicht möglich ist.

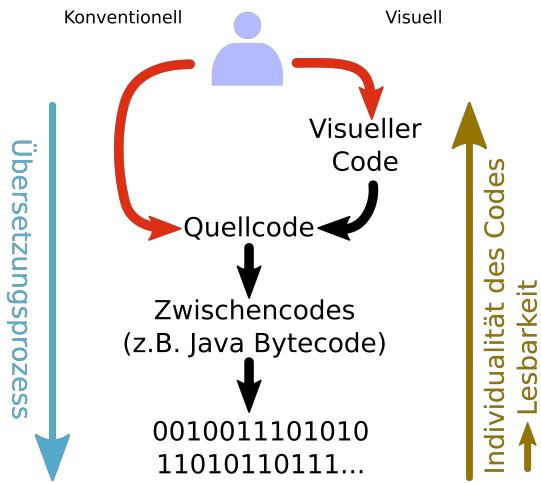


Abbildung 2.3: Übersetzungsablauf bei Programmierung

Die visuelle Programmierung setzt in aller Regel genau an der Schnittstelle zwischen Programmierer und Computer an, sodass der Programmierer nur noch mit der visuellen Programmiersprache in Kontakt steht, welche sowohl die Stärken des Programmierers, als auch die Funktionsweise des Computers berücksichtigt, mit dieser programmiert und der Rest automatisch passiert.

Allerdings gilt neben den durch die visuelle Programmierung entstehenden Vorteilen zu beachten, dass jedes mal, wenn ein Layer in eine Kette von verschiedenen Prozessen eingefügt wird, auch eine weitere potentielle Fehlerquelle entstanden ist. Es muss also davon ausgegangen werden, dass bei diesem neuen Layer auch zusätzliche Bugs und Sicherheitslücken entstehen. Außerdem ist es nicht nur Fluch, sondern auch Segen eine so minimalistische Informationsdarstellung wie Text als universellen Standard zu verwenden. Nichts ist nerviger, als Standartsalat, weshalb bei der Entwicklung dieser Systeme stets darauf geachtet werden sollte sich schnell auf sinnvolle Standards zu einigen.

2.3.1 Verknüpfung von Bildern zu Inhalt

Warum gibt es bei dem ursprünglichen Programmierer eines Scripts eine noch viel größere Geschwindigkeitsdifferenz als bei textuellen Scripts, Fehler schneller zu finden und zu verstehen gegenüber einem außenstehenden Programmierer (diese Annahme beruht auf Erfahrung von mir und einigen anderen Entwicklern, die sich damit befasst haben)? Auf der einen Seite existiert ein Quelltext den der Computer benötigt. Auf der Anderen haben wir im Besten Falle individuelle Bilder, die sich das Gehirn von Programmabschnitten gemerkt hat (Kontur, Farbe, Kontrast etc.). Wenn man einen visuellen Programmteil wieder sieht, kann das Gehirn z.B. schnell sagen 'das hab ich schon einmal gesehen, die Kontur kommt mir bekannt vor'. Diese Bilder haben zwar ohne weiteres nicht viel mit dem Inhalt zu tun, der visuell programmiert wurde, allerdings entsteht automatisch eine Art transitive Beziehung von den Bildern als Zugang für den Programmierer zum verfassten Inhalt (also quasi zu dem, was der Quelltext bedeutet) über den visuellen Code, weil der Programmierer beim visuellen Programmieren beides gleichzeitig verfasst hat. Dadurch kann das Gehirn des ursprünglichen Programmierers sogar direkt sagen 'das ich ich schon einmal gesehen und dort habe ich dies und jenes gemacht' [BILD] Für einen Programmierer, der ein fremdes Programmscript liest ist dieser Effekt natürlich nicht vorhanden, deshalb sollte die visuelle Repräsentation der Programminformationen so gewählt sein, dass visueller Code vom Gehirn intuitiv (also ohne "durchlesen") korrekt interpretiert wird, mehr dazu weiter unten beim Node-Based Visual Programming.

2.3.2 Spaß beim Programmieren

Wenn das Programmieren Spaß machen soll und der Programmierprozess schneller und effizienter werden soll (was untrennbar mit dem Spaß verbunden ist), dann muss das Design auch entsprechend ansprechend sein. Dies ist nicht bei allen existierenden visuellen Programmiersprachen der Fall, die es gibt. Ist dies nicht gegeben wird schnell das Ziel des Konzepts verfehlt, da meist wieder nur eine möglichst minimalistische

Darstellung von Information angestrebt wird. Es ist auch wichtig, dass der Programmierer das Design des visuellen Scripts bis zu einem bestimmten Grad selbst bestimmen kann (natürlich ohne dabei die Funktionalität zu verändern), jedoch gilt es dabei die im vorangegangenen Kapitel angesprochene Balance zwischen Individualität und Einheitlichkeit zu beachten.

2.3.3 Rahmenbedingungen für visuelle Programmiersprache

Um es zusammenzufassen hier einmal Rahmenbedingungen für eine moderne visuelle Programmiersprache aufgelistet.

Die Programmiersprache ...

1. ist visuell ansprechend
2. nutzt die Fähigkeit des Gehirns mit Graphiken umzugehen
3. besitzt bei der Repräsentation der Programminformationen eine gute Balance zwischen Individualität des Aussehens des Codes und Einheitlichkeit für die universelle Anwendung

2.4 Mögliche Funktionsweisen

Es gibt natürlich theoretisch viele Möglichkeiten, Quelltext sinnvoll graphisch darzustellen. Alle Versuche, sich von der Netzstruktur aus dem Node-Based Visual Programming zu entfernen sind bisher jedoch ohne Erfolg geblieben. Scratch behandle ich hier nicht, da es bei Scratch nur darum geht, die konventionelle, textbasierte Programmierung zu veranschaulichen und nicht, die Struktur des Programms selbst anders darzustellen.

2.4.1 Node-Based Visual Programming

Die bisher wohl am professionellsten umgesetzte Art der visuellen Programmierung ist das sog. „Node-Based Visual Programming“ („Knotenpunktbasierende visuelle Programmierung“). Dieser Name wird schnell verständlicher bei Folgendem Bild:

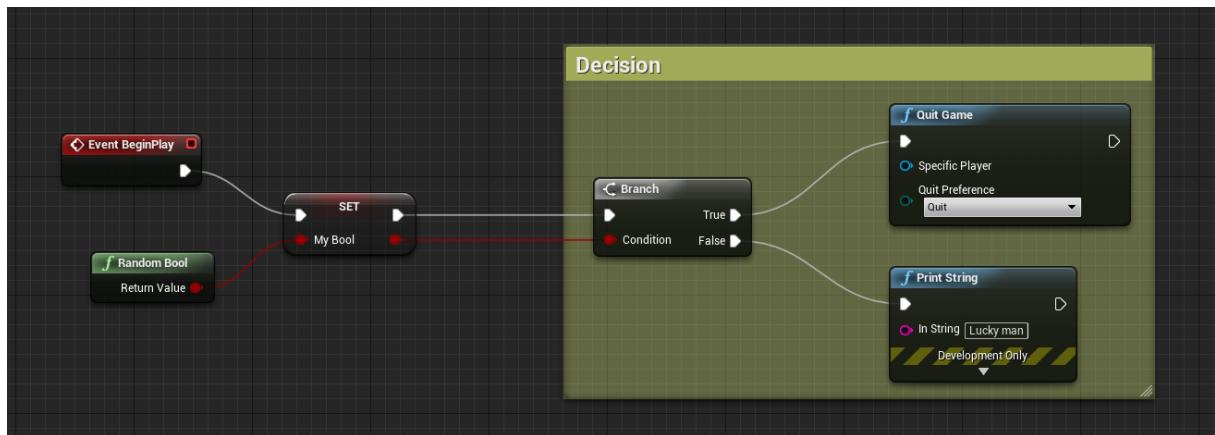


Abbildung 2.4: Visuelles Beispielscript - Node-Based Visual Programming

Hier gibt es Blöcke bzw. Knotenpunkte (Nodes) und zwischen diesen, Verbindungen. So entsteht ein zweidimensionales Netz von Operationen, denn jede Node steht für irgendeinen Code, bzw. eine Operation (in etwa, wie eine Anweisung im Quellcode).

Exkurs Das 'Node-Based' Repräsentierungsparadigma ist absichtlich dazu konzipiert eine Ablaufstruktur von Programmatik darzustellen, die der Denkweise des menschlichen Hirns mehr ähnelt, als einfacher Text (Schritt-für-Schritt-Denken mit Vernetzungspotential durch die zweidimensionale Darstellung). Ein Beispiel sind einfache mathematische Rechnungen. Um zu zeigen, dass dieses Repräsentierungsschema nicht nur bei konventioneller Programmierung sinnvoll angewendet werden

könnte und somit sehr skalierbar ist, schrieb ich als Zweitprojekt einen anderen Editor, in welchem es möglich ist einfache Mathematik-Simulationen zu erstellen (auch hier geht es nicht um ein professionelles Endprodukt, sondern um das Beweisen der realistischen Umsetzbarkeit an sich) (mat18).

2.4.2 Sonstige

Neben dem Node-Based Visual Programming gibt es leider noch nichts nennenswertes, was für diese Arbeit von Bedeutung ist.

2.4.3 Bekannte Beispiele

Es gibt bereits einige Beispiele für die erfolgreiche (und professionelle) Umsetzung einer visuellen Programmierumgebung (oder Programmiersprache). Es gibt natürlich noch mehr gute Systeme, die hier nicht aufgezählt sind, ich erwähne nur die größten.

BluePrints (Unreal Engine 4) Das oben gezeigte Bild zeigt die derzeit (mit Abstand) professionellste visuelle Programmiersprache, die auf dem Markt zu finden ist. BluePrints ist eine Engine-interne visuelle Programmiersprache der High-End-GameEngine Unreal Engine 4. Diese (einfach zu benutzende) visuelle Programmiersprache ist ein entscheidender Grund, warum die Engine heute für so vieles benutzt wird, obwohl sie eigentlich bis heute zu der High-End-Familie der GameEngines gehört. In der Unreal Engine werden verschiedenste Aufgabenbereiche (Programmscripts, Materials, Particle Systems, Animations und Sound-Scripts) mit diesem System weitgehend steuerbar gemacht. (ue418)

Godot ist auch eine GameEngine (allerdings aus dem Indie-Bereich), welche ebenfalls über ein visuelles Programmiersystem verfügt, das dem aus der Unreal Engine konzeptionell sehr ähnelt, aber etwas näher am Quellcode gehalten ist (god18).

NodeRed ist eine visuelle Programmiersprache, welche auf Node.js läuft, über einen normalen Internetbrowser benutzt werden kann um visuelle Scripte zu schreiben und Vernetzungen im Bereich Internet of Things einfach macht (nod18a).

Grasshopper ist eine visuelle Programmiersprache des 3D-Programms Rhino. Mit ihr ist es möglich algorithmisch 3D-Modeling zu betreiben und somit architektonische Bauwerke digital zu erstellen, welche ein, durch die Algorithmik generierbares, beeindruckendes Design haben (rhi18).

Houdini sollte dem Einen oder Anderen ein Begriff sein. Es ist eine professionelle, meist im High-End Filmgenre eingesetzte VFX-Software. In Houdini spielt auch eine Art visuelles Bausteinsystem eine ganz zentrale Rolle (die Effekte werden zu großem Teil nur mit diesen Blöcken erstellt und koordiniert). Allerdings wäre beim Design dieses Systems durchaus nachzuhelfen, ich halte es jedoch für möglich, dass die Entwicklerfirma der Software (SideFX) in der Zukunft ähnlich wirtschaften wird, wie EpicGames mit der Unreal Engine und sowohl durch die Kombination von einer High-End-Software, die durch ein Bausteinsystem aber einfach zu handhaben ist, als auch durch eine fast völlige kostenlose Verfügbarkeit den größten Teil des Marktes einnehmen wird (da die Schüler und Studenten, die in Zukunft Filme machen werden diese Software mit offenen Armen entgegen nehmen würden, genau wie es bei der Unreal Engine der Fall ist) (hou18).

Auffällig ist, dass sie alle ähnlich funktionieren, nämlich Node-Based. Es gibt natürlich noch mehr nützliche visuelle Programmiersprachen, diese sind dann aber meist für einen ganz bestimmten Anwendungsbereich oder sogar für eine ganz bestimmte Maschine und deshalb bei der Betrachtung der universellen Einsetzbarkeit nicht so wertvoll, wie die genannten, welche ein verhältnismäßig großes Funktionsspektrum aufweisen. Doch schon diese fünf Beispiele zeigen, dass visuelle Programmierung auch im professionellen Bereich großer Firmen wie SideFX oder EpicGames (Fortnite ist unter Anderem von EpicGames) durchaus von Nutzen sein kann.

In meinem Projekt geht es zwar grundsätzlich um eine universelle Einsetzbarkeit, dennoch sollte man nicht vergessen, dass die visuelle Programmierung auch bei dem aktuell in der Informationstechnik beobachtbaren Trend zu Domain Specific Languages (DSL) eine sehr nützliche Rolle spielen kann. Durch den hohen Grad an Problemspezifität kann man die Informationsdarstellung noch intuitiver gestalten, da nicht die gesamte Funktionalität einer großen Programmiersprache wie Java repräsentiert werden muss.

2.4.4 Fazit

[ich muss irgendwo noch sagen, dass der Hauptvorteil beim Debugging und Fehler vorbeugen besteht] Zum Einen sind manche Programmierer leider schlicht zu konservativ, sich mit etwas zu befassen, was nicht auf bereits erlernten Architekturen aufbaut, was die Entwicklung dieser Systeme hemmt, außerdem braucht auch der Umgang mit solch einer visuellen Darstellung der Programminformationen Übung, so wie man die braucht, um als völliger Neuling textuelle Programmierparadigmen zu verinnerlichen. Auch entsteht leider vereinzelt die Ansicht, die visuelle Programmierung generell sei bei manchen Programmierern nicht so zielführend, wie Text, weil man sich evtl. nicht sofort mit dem Node-Based Visual Programming - um welches es im Folgenden hauptsächlich gehen wird - identifizieren kann (anormale Menschen wie Autisten seien hier natürlich ausgenommen). Denn mit visueller Programmierung sei hier alles eingeschlossen, was die Programminformationen mit graphischen Elementen repräsentiert, welche sich nicht auf Text beschränken. Das beschränkt sich nicht auf Node-Based Visual Programming, sondern in erster Linie darauf, dass der Programmierer durch biologische Veranlagung intuitiver damit umgehen kann, wogegen ohne Spezifizierung auf eine spezielle Umsetzung in der Realität nichts einzuwenden sein sollte (wobei es da natürlich haufenweise Probleme geben kann).

3 Umsetzung

3.1 Motivation

Die prinzipielle Motivation wurde bereits erläutert. Hintergrundgedanke war: wenn ich es schaffe, die Rahmenbedingungen einigermaßen zu erfüllen, dann schafft das eine professionelle Entwicklerfirma in professionell mit links.

3.2 Node-Based Visual Programming im Detail

Geschriebenen Programmcode kann man prinzipiell folgendermaßen vereinfachen: Es gibt (von der Programmiersprache vordefinierte) Aufrufe oder Anweisungen (auch, wenn es sich nur um eine Zuweisung eines Wertes auf eine Variable handelt). Außerdem weiß ein Programmscript immer eine bestimmte Struktur auf, also die Festlegung was wann (und abhängig von was), wie oft usw. passiert. Um den Ablauf des Programms zu koordinieren und die vom Computer im Rahmen dieser Aufrufe ablaufenden Prozeduren zu spezifizieren, nutzen wir Daten (Werte, Ausdrücke etc.).

Eine 2D Netzstruktur zielt auf genau die genannten Vorteile der menschlichen Stärke der graphischen Strukturerkennung ab. Netz: bedeutet im weitesten Sinne, es existieren Verbindungen, die hier und dort übereinander laufen und Knotenpunkte bilden, welche in diesem Fall irgendwelche Informationen repräsentieren. Man beachte, dass Mindmaps einen äquivalenten Aufbau verfolgen. Dass Mindmaps mit etwas Übung sehr gut funktionieren ist kein Geheimnis, sie zielen genau auf das gleiche ab, wie die visuelle Programmierung. Nun kann man definieren, dass ein Knotenpunkt für eine bestimmte Aktion steht und eine Verbindung für das ‚Verbinden‘ dieser Aktionen untereinander, bzw. für das festlegen der oben beschriebenen Struktur. Achtung: dies muss nicht so sein, es ist durchaus möglich Programmcode anders und trotzdem netzartig darzustellen, ich wähle hier allerdings diese Methode, da sie einfach und naheliegend ist und zudem den Standard bei visueller Programmierung abbildet.

So eine Netzstruktur würde also irgendwie so aussehen:



Abbildung 3.1: Vereinfachter Programmablauf

Nun könnte einem aufgefallen sein, dass dies quasi einer ausführlichen horizontalen Visualisierung des AST ('Abstract Syntax Tree'), welcher ein Programm universell beschreibt, entspricht. Über AST und Mindmaps hinaus, erinnert die Darstellung vom Ablauf her auch an Flussdiagramme, Zustandsdiagramme und ein wenig an Struktogramme (auch wenn diese in der Realität individuell und in der Folge auch oft nicht schön lesbar sind, da die Richtung nach der intuitiven Interpretation des Gehirns, dass z.B. jeder Block für eine Aktion/einen Status/eine Maschine bzw. irgendeine in sich geschlossene Einheit steht, fehlt). Beim Node-Based Visual Programming richtet man sich also nach Diagrammtypen, die der übersichtlichen, leicht verständlichen und universellen Beschreibung von Programmen dienen.

Auch hier sind die Übergänge von einzelnen Aktionen, anders als bei Struktogrammen, durch dynamische

Verbindungslien dargestellt, was es dem Nutzer ermöglicht die Position der einzelnen Blöcke zu verändern und somit ein nicht einheitliches (Kontur-)Bild für einen bestimmten Programmteil zu generieren, was dem Gehirn dabei hilft die Stelle später wiederzuerkennen, weil sie nicht intuitiv genauso aussieht, wie jede andere im Programm.

Mit der Visualisierung des Programmcodes, wie oben gezeigt hat man zwar bereits den wichtigsten Vorteil dieser Technik implementiert (die Nutzung der Bilderkennungsfähigkeiten des menschlichen Hirns), jedoch ist dieser Effekt erst dann deutlich erkennbar, wenn das System durch Features erweitert wird, die diese Netzstruktur unterstützen.

Oben war von Daten die Rede und eine Möglichkeit, den Datentransfer zwischen verschiedenen Anweisungen (was genau damit gemeint ist folgt in Kürze) in die graphische Darstellung des Programmcodes so zu integrieren - sodass er die Netzstruktur unterstützt - ist, diesen Transfer ebenfalls mithilfe von Verbindungslien darzustellen.

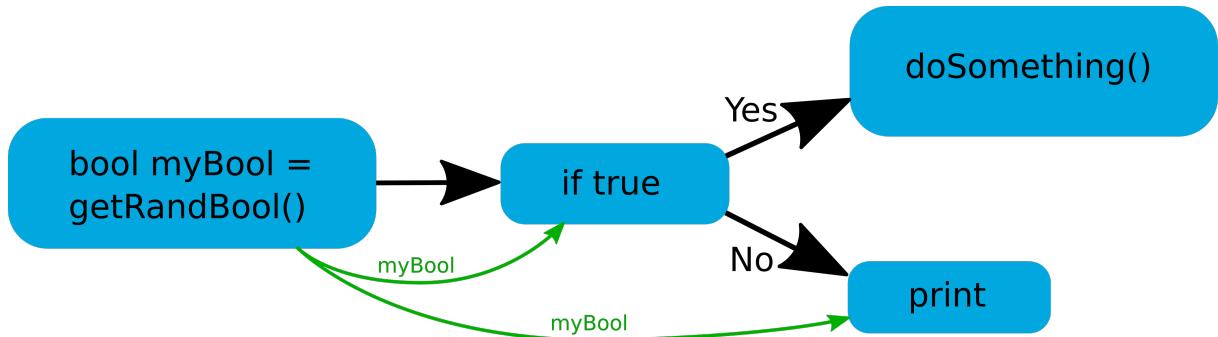


Abbildung 3.2: Vereinfachter Programmablauf 2

Im Quelltext sieht der Zugriff zwar anders aus (da man den Variablennamen benutzt und in keiner Weise auf die vorherige Zuweisung verlinkt), aber eine solche Darstellung verändert nicht den Sinn und macht den Programmteil verständlicher, was später bei dem Beispiel mit dem For-Loop noch deutlicher wird. Die Idee, dass man solch einen Datentransfer zwischen den Knoten generiert ermöglicht eine charakteristischere Darstellung und gleichzeitig bessere Übersichtlichkeit. Außerdem ist diese Implementation des Datentransfers bei mathematischen und logischen Ausdrücken sehr nützlich.

Hier hat die Graphik nun ein sehr eigenartiges Bild bekommen, welches sich das Gehirn problemlos merken kann. Natürlich merkt sich das Gehirn nicht den ganzen enthaltenen Text, aber die Anordnung der Blöcke, die Größenunterschiede, die charakteristischen Pfeile, all dies merkt sich das Gehirn erstaunlich gut. Somit ist also der Punkt erreicht, an dem ein Programm graphisch (also nicht mehr ausschließlich Textbasiert) dargestellt werden kann und die genannten Rahmenbedingungen grundlegend erfüllt (was nicht heißt, dass man die Darstellung nicht noch weiter mit ähnlichen Features signifikant verbessern kann).

3.3 Tools

Das Programm ist mithilfe der GUI-Bibliothek Qt programmiert. Benutzte Sprachen sind:

- C++ (nahezu 100% der Programmfunktionalität)
- QML (Markup-Language von Qt)
- JavaScript (im Rahmen von QML)
- qss (StyleSheet-Language von Qt)
- SQL (SQLite)

An dieser Stelle möchte ich mich kurz bei den aktiven Mitgliedern des Qt-Forums bedanken, welche mir oftmals trotz meiner Anfängerbeiträge (mittlerweile sind es insgesamt über 200 im Rahmen dieses Projekts) sich doch immer wieder Zeit genommen haben, mir bei der Lösung meiner Probleme zu helfen.

3.4 Überblick

Das Programm ist quasi ein Editor, aber als Standalone-Applikation. Ich habe das Programm also nicht dazu konzipiert als Plugin universell verwendbar zu sein, denn diese Art der Universalität war nicht das Ziel des Projekts. Der Aufbau ist den bereits vorgestellten Node-Based-Systemen nachempfunden. Ich habe mich dabei maßgeblich an dem visuellen Programmiersystem aus der Unreal Engine orientiert. Zusammengefasste Key-Elemente:

Graph Im Graph wird das visuelle Script mit den Nodes und ihren Connections dargestellt und editiert. Er dient also als „Anzeigefläche“

Nodes Es existieren Nodes (Knotenpunkte), wobei jede Node für eine bestimmte Operation, also einen Befehl oder eine Kontrollstruktur o.ä. steht (also im oben gezeigten Schema die abgerundeten Blöcke). Eine Node kann es nur einmal geben. Es kann von dieser Node aber mehrere NodeInstances geben. Man will z.B., dass das Programm eine If-Node bzw. ihre Daten (Titel, Beschreibung, Code etc.) kennt. Diese If-Node möchte man aber mehrmals platzieren/benutzen, also werden GUI-Instanzen (NodeInstances) dieser Node generiert. Die Node selbst wird dabei natürlich nirgendwo gezeichnet, sie dient nur als Schablone für ihre Instanzen.

Connections Die dargestellten Nodes im Graph werden mit Connections miteinander verbunden.

Execution-Connections sind die im Schema schwarz eingezeichneten Pfeile (im Programm nur Linien, da sie nur eine mögliche Laufrichtung haben). Sie regeln den zeitlichen Ablauf. Man kann sie sich also quasi als Impulsübergeber vorstellen.

Data-Connections sind die farbigen Pfeile (im Programm auch Linien). Sie übermitteln Daten, also Werte, Variablen oder Ausdrücke.

Grundsätzlich kann man die Klassenstruktur (soweit sie hier von Bedeutung ist) wie folgt darstellen:

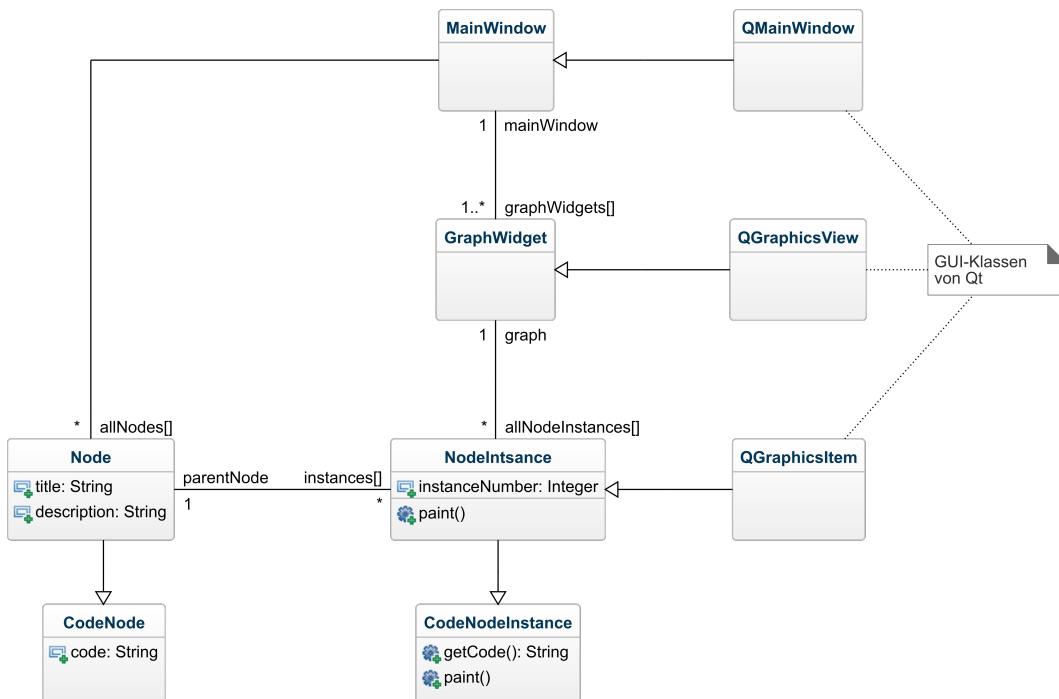


Abbildung 3.3: Klassenstruktur

Das zentrale Element der Nodes für die Codegenerierung (der ‚Metacode‘) liegt als das Attribut ‚code‘ in der **CodeNode**-Klasse vor. Es gibt potentiell noch andere Node-Arten, die hier aber nicht näher von Bedeutung sind, deshalb geht es hier nur um die **CodeNodes**.

Ich habe in meinem Programm die Darstellungsart so gewählt, dass das oben gezeigte Schema eher solch ein Design haben würde:

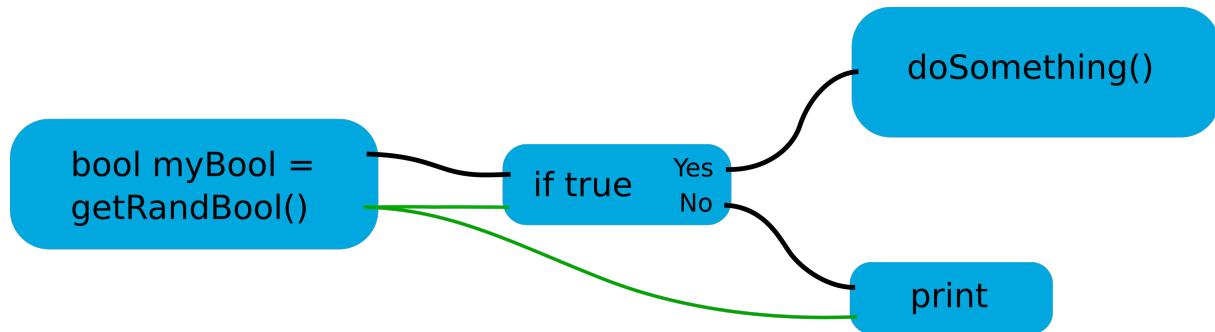


Abbildung 3.4: Vereinfachter Programmablauf 3

3.5 Funktionsweise des Key-Features Codegenerierung

Es stellte sich als am Zielführendsten heraus, das graphische Script genau so auszuwerten, wie ein Mensch es liest, ohne große Konvertierungen aus dieser (zweidimensionalen) Darstellungsart heraus. Der Metacode sagt dem Programm, bei Generierung des gesamten Projektcodes quasi, wie genau der Code für die entsprechende Node zu generieren ist. Alles, was wir in einer Programmiersprache schreiben ist an eine bestimmte Syntax geknüpft. Teile dieser Syntax sind fest, andere Teile können (und sollen) wir (mehr oder weniger) frei bestimmen. Die frei bestimmbaren Teile der Syntax legen wir im graphischen Script fest durch die miteinander verbundenen Nodes und das, was wir in ihre Inputs schreiben (siehe Beispiele). Die nicht änderbaren Teile soll der Programmierer bei der visuellen Programmierung in der Regel (hier auch) nicht mehr selbst schreiben müssen. Dort zieht man die Blöcke einfach in das Script und ändert die Dinge, die man ändern darf. Ich habe das hier genauso implementiert.

Ganz vereinfacht kann man den Codegenerierungsalgorithmus (also die getCode()-Funktion in CodeNodeInstance.cpp, siehe Abbildung 3.3) folgendermaßen darstellen:

```

string mainCode = parentNode.code # hier ist es noch 'Meta-Code'
# alle %INPUTX%-Notationen im mainCode austauschen lassen
inputNotation = regularExpression('%INPUT\d%')
for inputNotation in mainCode:
    if currentInput connected: # mit einem anderen Output
        mainCode.replace(currentInputNotation,
                         connectedOutput.getCode(currentInput))
    else:
        mainCode.replace(currentInputNotation,
                         currentInput.widget.getDataString())
# alle %OUTPUTX%-Notationen im mainCode austauschen lassen
outputNotation = regularExpression('%OUTPUT\d%')
for outputNotation in mainCode:
    if currentOutput connected: # mit einem anderen Input
        mainCode.replace(currentOutputNotation,
                         connectedNodeInstance.getCode() #* Rekursion *#)
        """
        im mainCode dürfen nur Output-Notationen liegen, die auf
        Execution-Outputs verweisen, Data-Outputs werden in
        getCode verarbeitet """
    else:
        mainCode.replace(currentOutputNotation, NEWLINE)
return mainCode
  
```

Es werden also eigentlich nur die entsprechenden Notationen ersetzt. Allerdings ist diese Art der Codegenerierung noch etwas primitiv und eingeschränkt. Es kann z.B. nicht gesagt werden 'platziere das else nur bei verbundenem else-Output', denn das wäre spezifische Funktionalität wofür Programmierung benötigt wird, weshalb ich plane eine DSL (Domain Specific Language) für den Metacode zu schreiben in welcher solche, aber auch etwas weiterreichende Befehle implementiert sind, was dem Programmierer auch die Möglichkeit gibt den entstehenden Code so zu organisieren, dass er besser lesbar ist. Der Algorithmus wird verständlicher, bei den folgenden Beispielen. Er ist zwar eigentlich deutlich größer, aber Mechanismen, wie das automatische Ergänzen der Einrückung der aktuellen Zeile in alle Zeilen, die von einer anderen NodeInstance von getCode() zurückgegeben werden, sind hier nicht von zentraler Bedeutung. Prinzipiell kann man die Funktionsweisen der Inputs und Outputs folgendermaßen erklären:

Execution-Inputs dienen als einfacher Aufruf, also Festlegung, welche Node. nach welcher ausgeführt werden soll

Data-Inputs repräsentieren bei den Nodes quasi einzelne kleine eingebettete Stellen im Quellcode (z.B. die Abfrage im Head einer Verzweigung)

Execution-Outputs repräsentieren immer ganze Code-Blöcke (z.B. den Body . eines For-Loops). Das heißt, bei jedem Execution-Output weiß ich, die hier angeschlossene Node kommt entweder ganz nach der Node auf die ich mich beziehe oder bildet einen Code-Block ab, der Teil der Node ist auf die ich mich beziehe. Verständlicher wird dieses System beim betrachten der Meta-Codes von Kontrollstrukturen.

Data-Outputs dienen als direkte Verlinkungen auf Daten (also Variablen oder ganze Ausdrücke). Dies ist nicht unbedingt nötig, bietet aber eine gute Möglichkeit das visuelle Script anschaulicher zu gestalten (wie oben beschrieben). Bei einem Data-Output muss also klar angegeben sein, was dieser Data-Output repräsentiert (z.B. die Zählvariable bei einem For-Loop, oder im Schema oben eine Variable, der gerade ein neuer Wert zugewiesen wurde). Dies ist ein Paradebeispiel dafür, wie man mit visueller Programmierung nicht nur Quellcode 1:1 repräsentieren kann, sondern durch die zusätzlichen Möglichkeiten auch Zusatzfeatures implementieren kann.

3.5.1 Beispiel für normale Anweisungen

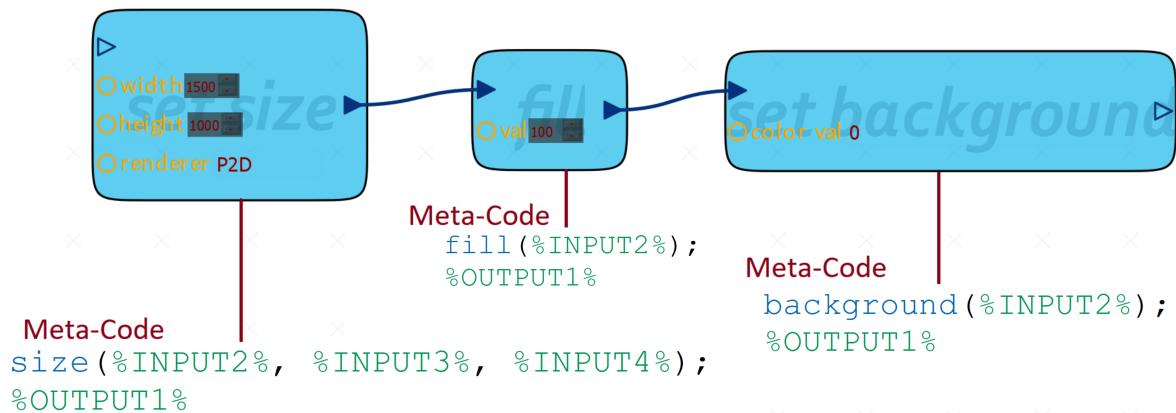


Abbildung 3.5: Beispiel für normale Anweisungen

(im Programm sind nur die Blöcke, also die Nodes und die Verbindungen untereinander sichtbar, nicht die Meta-Code-Kästen, diese dienen hier nur als Veranschaulichung)

Der Code-Generierungsalgorithmus generiert hier folgenden Quelltext:

```

size(1500, 1000, P2D);
fill(100);
background(0);

```

Abbildung 3.6: Beispiel für normale Anweisungen - generierter Code

Erläuterung Die %INPUTX%-und %OUTPUTX%-Notationen dienen als Anker für Reguläre Ausdrücke (Regular Expressions) immer ausgehend von einer bestimmten NodeInstance. Dies schränkt natürlich die Darstellbarkeit von Text ein, da der resultierende Code keine dieser Notationen mehr enthalten kann. Da Variablen- und Funktionsnamen in heutigen Programmiersprachen in der Regel nicht so aussehen dürfen, entschied ich mich für diese (einfache) Lösung. Es wäre aber auch möglich separat für jede Node zu speichern, an welchen Stellen (in ihrem Meta-Code) sich welche Input- oder Output-Notationen befinden, um diesem Problem aus dem Weg zu gehen.
Der Algorithmus funktioniert quasi rekursiv, er ruft sich selbst wieder auf. Zuerst sucht sich die SetSize-NodeInstance (sie muss vor dem Codegenerieren angeklickt werden) alle %INPUT% im Meta-Code ihrer parentNode (siehe Attribut ‚code‘ in Klasse CodeNode in Klassenstruktur 3.3) heraus. Dann ersetzt sie die Aufkommen entweder mit den an dem entsprechenden Input verbundenen Daten (hier sind die Inputs nicht verbunden) oder durch die in die Widgets geschriebenen Daten.

Die SetSize-NodeInstance fragt, wenn sie das %OUTPUT1% in ihrem Meta-Code entdeckt nach dem vollständigen Code der Fill-NodeInstance (der Code-Generierungsalgorithmus von der SetSize-NodeInstance ruft also den der Fill-Node auf usw.). Wenn die SetSize-NodeInstance den Code von der Fill-NodeInstance zurück bekommt, ersetzt sie ihr %OUTPUT1% des Meta-Codes mit diesem, hat damit ihren Code vollständig und gibt ihn ihrerseits an denjenigen zurück, der gefragt hat (in dem Fall der User, könnte auch eine andere NodeInstance sein).

3.5.2 Vollständiges Beispiel

Um das Geschriebene nun zu vernetzen, hier ein vollständiges Beispiel: Folgender visueller code

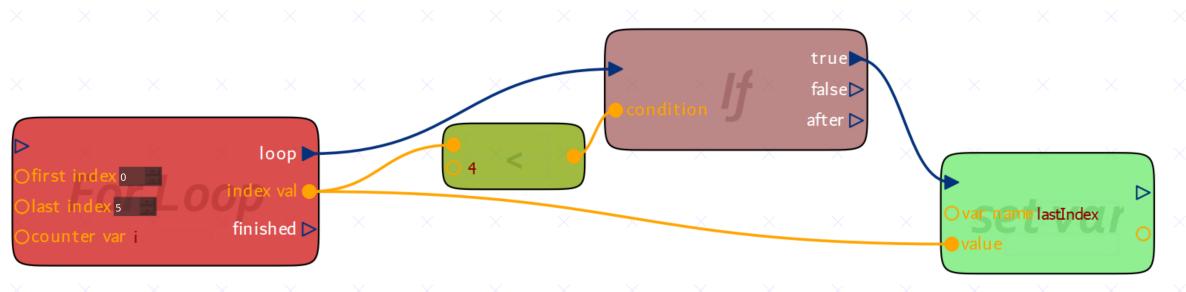


Abbildung 3.7: Beispiel für normale Anweisungen

generiert diesen Quellcode:

```

for(int i = 1; i <= 5; i++) {
    if(i < 4) {
        lastIndex = i;
    } else{
    }
}

```

Abbildung 3.8: Beispiel für normale Anweisungen

Folgendes Sequenzdiagramm verdeutlicht den (leicht vereinfachten) zeitlichen Ablauf des Codegenerierungsalgorithmus' des dargestellten visuellen Scripts:

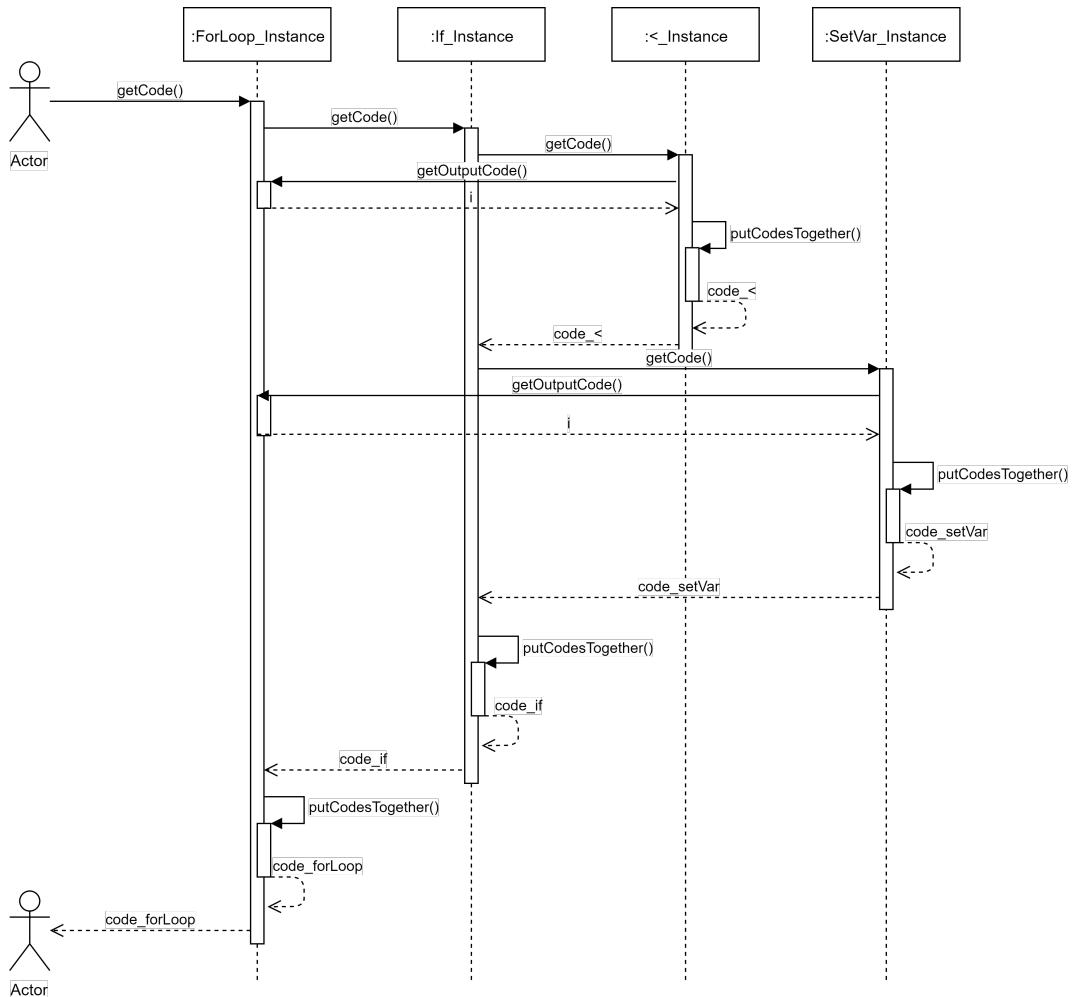


Abbildung 3.9: Codegenerierung Beispiel Sequenzdiagramm

3.6 Skalierbarkeit

Die Skalierbarkeit ist das Kriterium, welches mich wiederholt sehr weit in der Entwicklung dieses Systems zurückgeworfen hat. Da sowohl der Metacode der CodeNodes, als auch der Code aus den Data-Outputs reine Strings sind, gibt es hier bis auf die Input-und Output-Notationen faktisch keine Limitierungen (mit Verwendung einer geeigneten DSL wie beschrieben, fiele auch diese Einschränkung weg). In diesem Kapitel habe ich als Beispiel von mir erstellte Nodes benutzt, welche Java-Quelltext repräsentieren. Ich habe ein zweites Programm geschrieben, in welchem man Nodes mit allen dynamischen Attributen der Nodes (Meta-Code, Farbe, Titel, Beschreibung usw.) individuell erstellen kann. Diese werden von dort aus in Datenbanken abgespeichert und können anschließend in den Editor (das Hauptprogramm) geladen werden. Eine Datenbank beinhaltet dabei beliebig viele Nodes. Es ist jedoch so gedacht, dass eine Datenbank beispielsweise die gesamte Grundfunktionalität einer Programmiersprache beinhaltet (Bibliotheken können dann in weitere Datenbanken geschrieben werden). D.h. ich kann neben der Java-DB auch eine Python-DB erstellen, um im Editor in Python programmieren zu können.

Der oben gezeigte For-Loop sieht in der Python-DB so aus:

```
for %INPUT2% in range(%INPUT3%+1) :  
    %OUTPUT1%  
    %OUTPUT2%
```

Abbildung 3.10: Der Meta-Code für einen ForLoop für Python

Während der Skalierbarkeit keine Grenzen gesetzt sind, sind der Sinnhaftigkeit einzelner Darstellungen natürlich immer noch Grenzen gesetzt. So kann man z.B. das Definieren von Funktionen im Node-Based Visual Script auf verschiedene Weisen implementieren kann, genauso Klassen, Structs, Makros usw. Nicht alle möglichen Realisierungen machen Sinn. Es macht beispielsweise überhaupt keinen Sinn für das Definieren einer Funktion einfach eine Node zu erstellen, bei der ich den gesamten Quellcode der Funktion einfach in eine Input-LineEdit schreibe (wäre aber möglich). In BluePrints ist bereits das ein oder andere Feature (z.B. visuelle Kommentare) implementiert, das den Programmierprozess weiter unterstützt. In meiner Implementation existieren auch sog. Makros, also Nodes, die ihrerseits wieder einen kompletten Graph mit Node-Instanzen (und allem was dazu gehört) enthalten. Ein Makro kann mehrmals platziert bzw. instanziert werden (es können also beliebig viele MacroNodeInstances einer MacroNode platziert werden), womit es wiederverwendbar ist, unabhängig von der Anzahl und Art der Inputs und Outputs. Beliebig große Programmteile lassen sich in solche Makros packen, was ein sehr wichtiges Feature ist. Weitergehend ist es natürlich theoretisch möglich, aus gegebenem Quellcode und gegebenen Nodes mit Meta-Code den visuellen Code zu generieren. Jedoch können dann die durch die visuelle Darstellung ermöglichten zusätzlichen Features nicht oder nur spekulativ berücksichtigt werden. Es ist die Idee, dass der Programmierer sich bereits beim Programmierprozess die Kontur der Blöcke merkt.

4 Fazit und Ausblick

Wie beschrieben, bestehen schon ein paar sehr brauchbare Systeme, allerdings nur sehr wenige, obwohl so eine visuelle Programmierumgebung (zumindest für imperative Programmierung) nicht schwer zu schreiben ist und diese sind auch nicht universell einsetzbar. Dennoch lässt die Zukunft hoffen. Die BluePrints sind erst seit wenigen Jahren so gut, entwickeln sich jedoch mit rasanter Geschwindigkeit. Node-Red ist erst seit 2016 Open-Source (IBM hat in dem Jahr die Software der JS-Foundation übertragen (nod18b)). Und wenn man sich im Genre der Spielentwicklung ein wenig umhört, bekommt man doch immer wieder mit, wie beeindruckt viele Spieleentwickler (sowohl Hobbyentwickler, als auch professionelle Entwicklerstudios) von den BluePrints aus der Unreal Engine sind, was ein zentraler Grund dafür ist, dass diese - eigentlich für High-End-Gaming konzipierte - Engine inzwischen auch für ganz andere industrielle Bereiche (wie Architektur und Automobilindustrie) genutzt wird. Das kanadische Entwicklerstudio Behaviour Interactive Studios antwortete auf Anfrage, wie sehr bei ihrem Computerspiel Dead By Daylight von BluePrints Gebrauch gemacht wurde, es habe eine etwa gleiche Verteilung zwischen C++-Code und BluePrints gegeben. Die Designer, welche in der Regel über keine professionellen Programmierkenntnisse verfügen, hätten aber sehr davon profitiert, mit BluePrints alles ihre Arbeit betreffende einfach und übersichtlich steuern zu können.

Die Rahmenbedingungen für eine sinnvolle visuelle Programmiersprache zu erfüllen stellt kein durch technische Limitierungen eingeschränktes Problem dar, was die universelle Anwendbarkeit der von mir im Rahmen dieses Projektes geschriebenen Software zeigt. Auch durch moderne Wege der Programmierung über Runtime-Interpreter eröffnen sich bei intuitiven, den Programmierer unterstützenden, Umgebungen in Kombination mit visueller Programmierung gute Möglichkeiten (z.B. vollkommen dynamische Funktionalitäten von Nodes).

Literaturverzeichnis

- [god18] *Godot Engine*. https://docs.godotengine.org/en/3.0/getting_started/scripting/visual_script/getting_started.html, 2018
- [hou18] *Houdini*. <https://www.sidefx.com/products/houdini/>, 2018
- [mat18] *MathGraph*. <https://www.youtube.com/watch?v=iqhNd1lwJUk/>, 2018
- [nod18a] *NodeRed*. <https://nodered.org/>, 2018
- [nod18b] *NodeRed-Wikipedia*. <https://de.wikipedia.org/wiki/Node-RED>, 2018
- [rhi18] *Rhino - Grasshopper*. <https://www.rhino3d.com/6/new/grasshopper>, 2018
- [ue418] *EpicGames*. <https://docs.unrealengine.com/en-us/Engine/Blueprints>, 2018