Aleksandr Miroliubov

Visual Programming – An Alternative Way Of Developing Software

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Thesis

27 March 2018

Author Title Number of Pages Date	Aleksandr Miroliubov Visual Programming – An Alternative Way Of Developing Software 31 pages + 1 appendix 27 March 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Principal lecturer Erja Nikunen Senior Lecturer Jonita Martelius

As the society moves deeper into the Digital Age, the software requirements and the interaction between the client and the developer changes, affecting how to need to develop software and what the end result should be.

Visual programming is a tool well suited for smaller developer teams and in some cases even end users. The objective of this thesis was to familiarize myself with various visual programming languages, find out for what kind of software development they are suitable for and what their limits are.

In the thesis, I explain what visual programming is and how it works on a general level, as well as go through a few of the commonly used visual programming languages and in what kind of environments they are used in. Additionally, I explore the strengths and weaknesses of visual programming languages, in the hope of shining a light on some misconceptions people have about them.

In addition to my own work experience, I provide multiple studies in favor of visual programming, but also outline a few fundamental flaws that need to be addressed before visual programming can become truly mainstream.

Keywords	Visual programming, scripting, dataflow, software development

Contents

List of Abbreviations

VPL	Visual programming language. A programming language where the programmer manipulates graphical elements instead of text.
VS	Visual Scripting. The act of programming with visual expressions. It can be based on a text-based language with a visual representation of the program elements.
UE	Unreal Engine. Game development engine and environment.
FSM	Finite-state machine. A mathematical model of computation which is defined by a list of states and transition conditions.
API	Application Programming Interface. A set of subroutine definitions, protocols and tools for building application software.
AI	Artificial Intelligence. The capability of a machine to imitate intelligent behavior, typically in the form of problem solving.
LabVIEW	Laboratory Virtual Instrumentation Engineering Workbench. It is one of the most widely used visual data flow programming languages.

1 Introduction

Computers and the software they run are a big part of the modern society and they continue becoming an even bigger part of our lives. It's important to keep up with the way of thinking it requires to not only create them, but to interact with them as well. A lot of the time, communicating a complex thought or an idea is easiest in the form of various diagrams. Visual programming takes advantage of this to create shortcuts in design time, programming complexity and maintenance.

For example, if you want to set up a company in the modern society, in addition to the service or product the company sells, it requires an infrastructure; payment or online store systems, social networking for brand awareness and advertisement, website creation and other base structures. However, a small startup company can't necessarily afford to hire experts to create those things for them. This is one of the areas where visual programming simplifies the process. There are dozens of tools available to the user, ranging from website and media creation, to software creation and database systems. Visual programming provides a fairly simple solution to an otherwise complex problem and even a small investment goes a long way, as it requires considerably fewer programming concepts and less abstract thinking.

In the thesis I'll go through what visual programming is, what advantage it has compared to text-based programming languages and explain some of the reasons why it's not always used instead of text-based programming languages. In addition, I explore in what fields it's commonly used in, as well as make a simple program with a VPL, which I haven't used before.

1.1 What is visual programming

Visual programming is a style of programming where the user utilizes graphical elements, which represent functions, operators or variables, and connects them typically via lines or arrows, forming relations. VPL's can be classified into icon based, diagram based and form based languages. Icon and form based VPL's aren't very common in the present day due to the syntax restrictions of the languages, many of which were developed in the late 70s or 80s during the time when programmers were experimenting with visual programming.

Out of the three types, the diagram based languages are the most commonly used, especially those based on dataflow and state machines. They can be flexible and are relatively easy to read and understand, even by the end users. Examples of these are VPL's such as Drakon, Simulink and Bubble.

Visual programming can also be done on a more traditional programming language with a graphical representation of the elements. This includes some languages of Microsoft Visual Studio and game development environments such as UE and Unity. In these cases a compiler converts the visual representation of the program into code. Appendix 1 shows a movement template graph from UE and what the same code looks like when it's nativized to C++.

1.2 Data flow programming

As the name suggests, in data flow programming the execute order follows the flow of data along the input and output of the nodes, which are connected to each other via lines. The nodes themselves are functions or sometimes variables. By changing the order of the nodes or how they are connected, the user can change the flow of data. Example of a data flow based VPL is seen in figure 1.

Data flow programming is used due its fairly simplistic style, while maintaining the functionality. The less experienced user can manipulate the elements with the black box model in mind, where he doesn't need to know the inner workings of the function. It is enough to know what kind of variable goes into input and what comes from the output. In most data flow based VPL's you can't connect an incompatible variable into the input, which decreases user error. Most visual data flow programming environments provide instant visual feedback in case of incorrectly connected nodes or mismatch in input data type.

Data flow model has two approaches for executing the program. First one is the data driven approach, in which the execution order is dependent on the availability of the data, illustrated in figure 2. Second one is the demand driven approach. It executes instructions only as needed and starts from the final outputs of the program, requesting data from only the nodes it’s attached to. This weaves a network of only necessary instructions.

Another major feature is task parallelism, which requires separate libraries or API’s in imperative programming languages such as C++, where the user would need to use a library such as TBB or PASL. Programming languages where the programmer controls the flow of data are called imperative or control-flow languages and are based on the von Neumann model. In data flow programming the operation can be executed as soon as the required input nodes are known, this means that many operations can be run in parallel as long as they are not dependent on each other. For example the popular LabVIEW VPL has inherent parallelism system, which allows the execution of the program in multiple threads. In most imperative programming languages the programmer would need to explicitly create and handle threads. Parallel computing has become increasingly important in the last 10 years, since the multicore processors have become the norm [1].Task parallelism in data driven approach of the data flow model introduces a problem in unnecessary task execution, as it executes available instructions as soon as the data is available, even though the data might not be needed anywhere in the present scenario. This problem is corrected by the previously mentioned demand driven approach, which maps out a web starting from the end point and traversing towards the outer nodes only via the operations it requires. In figure 1, the start point of demand driven approach would be the LED and the end points would be the min and max values outside the for-loop.

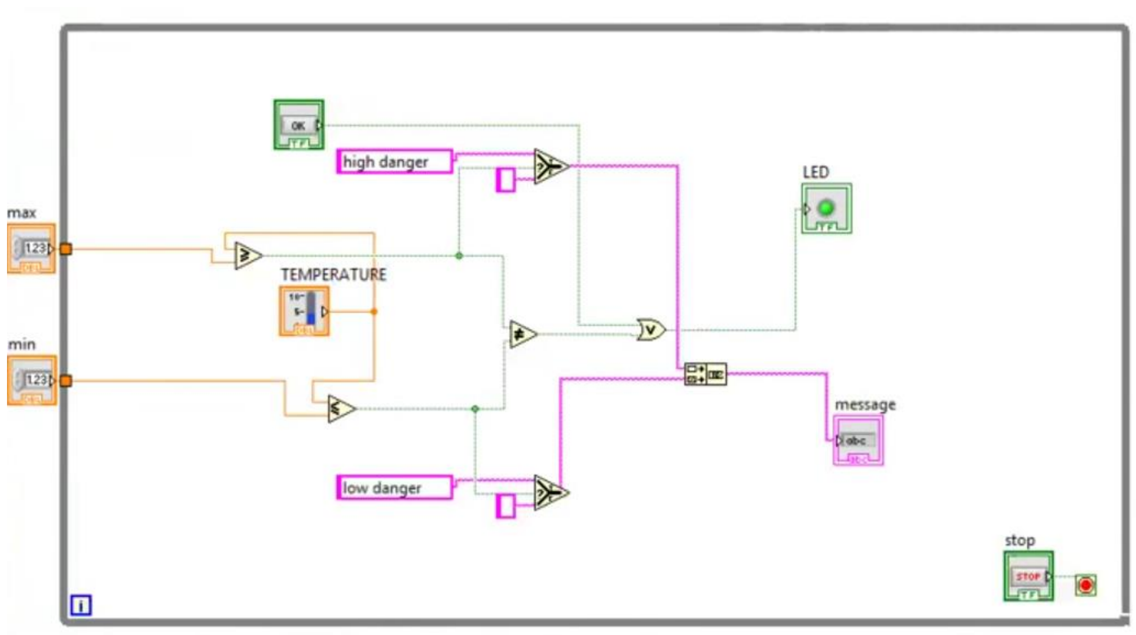


Figure 1. Data flow programming paradigm based VPL called LabVIEW, developed by National Instruments

Figure 1 is a fairly simple temperature control system made in LABView. It starts out from the left by assigning the maximum and minimum allowed temperatures which enter a for-loop, indicated by the gray box. The red icon at the bottom right is the loop end-condition, which is connected to a button, controlled by the user. The minimum and maximum numeric values, which are indicated by the orange wires, pass through comparison operators where the other input is the temperature value gathered from an electrical temperature sensor. The LED is activated if the signal reaches the node, which in this case will indicate temperatures over or under the assigned limits. Additionally, the select operators are used guide the string output, which by default is indicated by the pink line. This prints out a simple danger indication text. Right clicking the nodes will reveal additional options, which in the case of the LED node allows the user to change the on and off colors in addition to multiple other settings.

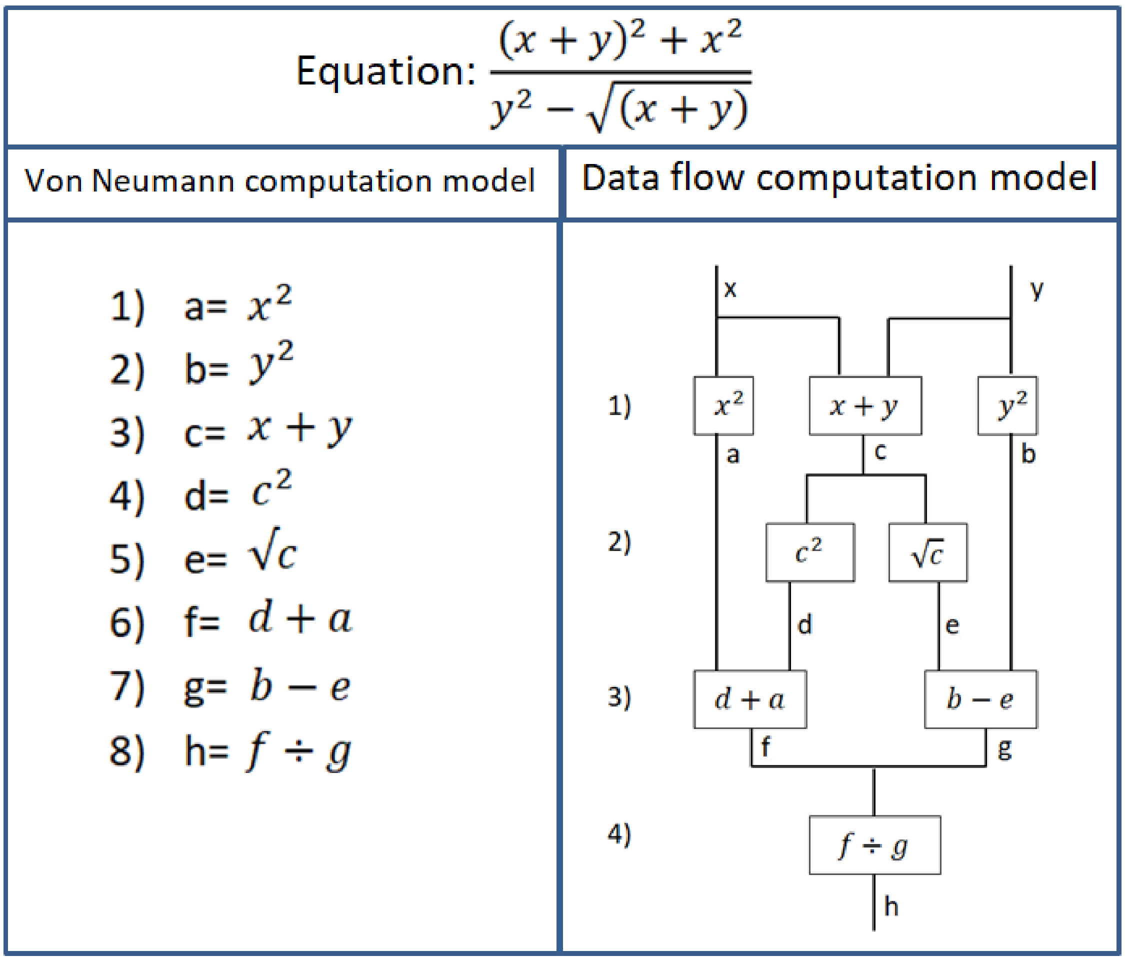


Figure 2. Von Neumann sequential computation model opposed to the data flow execution model.

Figure 2 examines how both von Neumann and the data flow execution models handle a simple computation task. One of the weaknesses of the von Neumann architecture, which most traditional programming languages are based on, is the sequential computation model, where a single instruction counter dictates which instruction is going to be executed next. Because of the single instruction counter, computing always proceeds sequentially and in a predefined order. This kind of computation model isn't taking advantage modern multi-core processors and the problem is commonly referred as the von Neumann bottleneck [12].In figure 2, the numbers on the side indicate the amount of steps taken towards completing the calculation. Data flow model allows multiple simultaneous calculations as long as all of the required variables are known. As seen in the figure, the first cycle computes three calculations because the variables required for the calculation are known, resulting tokens a, b and c. The same calculations would require 3 cycles using the von Neumann model. This bottleneck has become more problematic because increasing portion of the traffic consists of where to find the data, instead of the data itself.

1.3 Finite-state machines

While data flow based VPL’s are overwhelmingly more popular than FSM based, there are certain areas in which state machines are better, namely in reactive and event-driven systems. Many of the visual programming tools available in the statechart category are actually visual scripting languages as they convert the statecharts into classic programming languages. This is also the case with Yakindu, a state chart tool which features source code generators for Java, C and C++. Majority of these, feature the familiar drag and drop system with node connection and their conditions for transition, the same things you would see in basic UML state diagrams.

FSM’s are defined by its states and the conditions for the transition, which is the act of switching from one state to another. FSM offers many advantages such as simplicity, derivation and easy testing. Simplicity comes in the form of clear and easy to read charts with precise specifications, divided into states. However, manipulation of state change conditions still requires some basic programming knowledge. Derivation in this case is the easy transition from the clients’ requirements to the design and implementation phases. Testing an FSM is simple as test case scenarios can be generated from the FSM model directly [2].

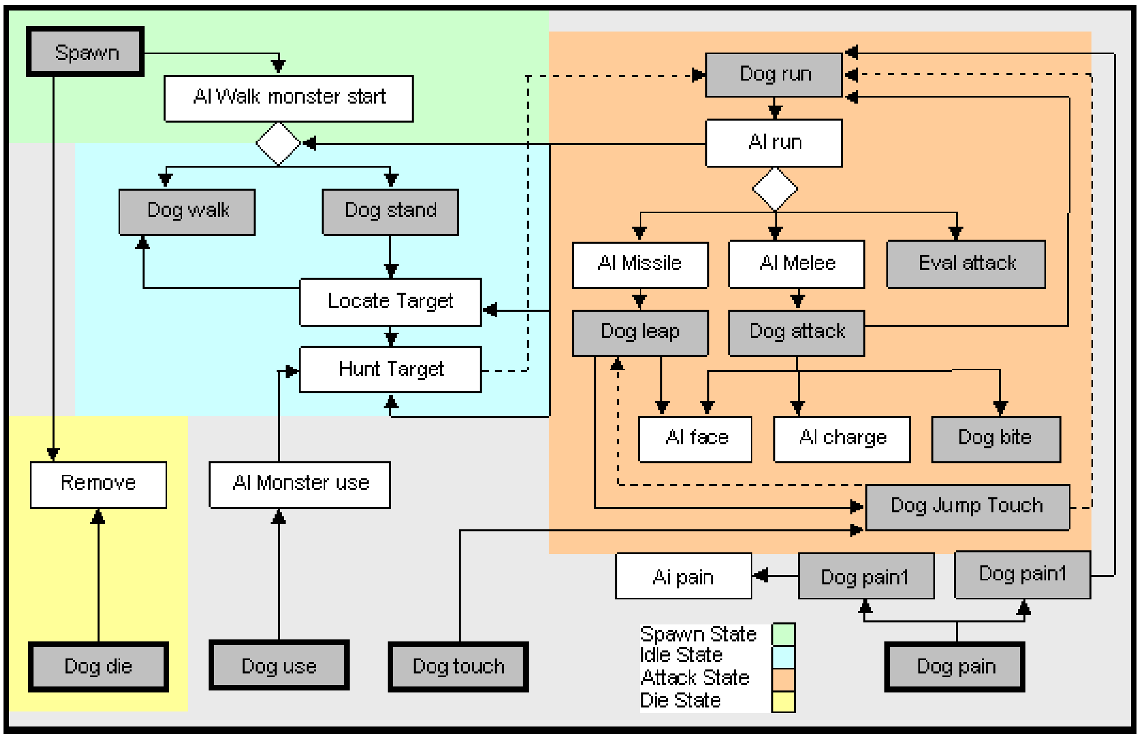


Figure 3. Dog monsters State Machine from Quake videogame.

While FSM can only be in one state at a time it usually loops back to the same state multiple times as FSM's are usually designed to be closed loops unlike in data flow programming. This makes state machines very well suited for various mechanisms, such as heaters, elevators and assembly line machinery. Finite state machines are commonly used in programming to outline how the completed program should work, however some visual scripting tools such as NodeCanvas and xaitControl can convert your state machine directly into source code. VPL's based on FSM are also commonly used in AI behavior programming [3]. Example of an AI State Machine can be seen in figure 3.

1.4 The roots of visual programming

While visual programming is not newly invented, it has had an increase of popularity since it has been introduced in education programs and game development engines such as Unreal Engine and Unity. In the past it's been used and experimented with as an education tool to teach kids logical thinking, but due to its many restrictions, it was deemed as too limited to be used instead of traditional text-based programming.

VPL’s had a rocky start in the 1970’s as the computer hardware couldn’t take the added strain of rendering images, boxes and arrows. Another problem was that panning and zooming was not yet common place and because the early OS were text based, so there was little use for a computer mouse. This made the early visual programming languages a tangled mess, which was hard to read and navigate, therefore making them inconvenient.

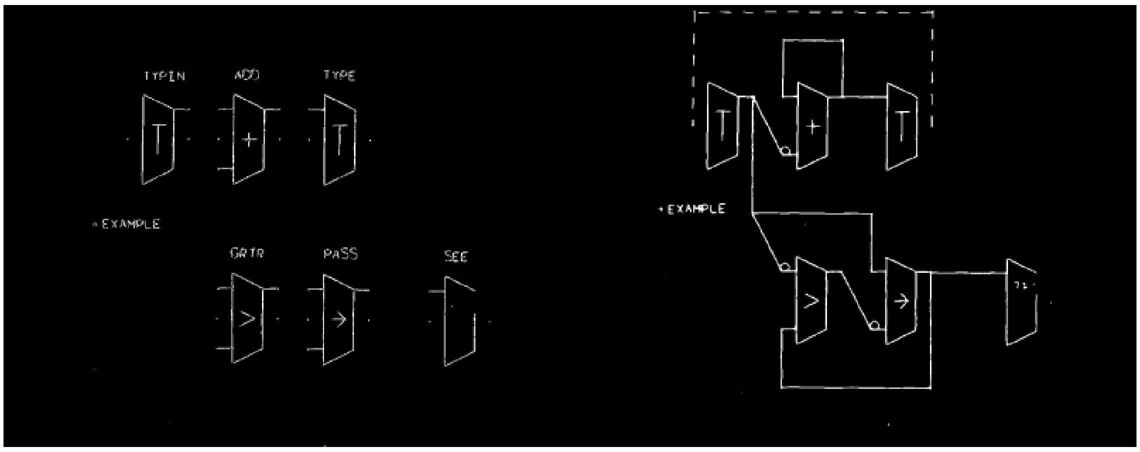


Figure 4. Basic operators and a program made by calling up replicas of the master symbols and then connecting the terminals with lines.

Arguably the first VPL was developed by William Sutherland in 1966 thesis for MIT department of Electrical Engineering, for which the author created a graphical programming system for the TX-2 computer, see figure 4. In this Visual programming system, the user could call arbitrary symbols and give them input values, after which the program could be executed. It also had the capability of combining operators into completely new symbols. The visual style of Sutherlands programming language resembled a logic diagram, where logic gates were swapped for graphical representation of operations [4].

2 Visual programming basics

While it’s much easier to understand how visual programming works compared to text-based, it still requires basic programming skills and knowledge how different elements interact with each other. A person with no prior programming skills would most likely not be able to perform any significant task, however while looking at the graphical elements, he would understand the general idea of what is happening, which is useful in larger teams consisting of members with varying expertise. In addition, the learning curve for visual programming languages is only a fraction compared to the text-based counterpart.

2.1 Variables and functions

Most of the variable type's used in traditional programming can be defined in visual programming languages, as illustrated in figure 5. This includes data types such as integer, float, Boolean and even arrays. Many modern VPL's allow the use of object variables, similarly used as in object-oriented programming languages such as Java, C++ and Python. Custom data types and similar data abstractions are a key in making a cleaner interface and for easier debugging and better scalability.

As mentioned before, most VPL's are based on the data flow model and in pure data flow languages, the variables can only be assigned a single time, so once the variable is created, it can't be modified. This means that pure data flow model isn't practical due to lack of control structures, such as iteration and condition. The upside of this is that with each output a new variable is created that is not dependent on any function, which is a key enabler for parallel computing. Visual data flow languages usually borrow elements from control-flow based languages, especially when it comes to global and local variables, even though it is not supported in pure data flow model. This ensures a certain level of flexibility in the code.

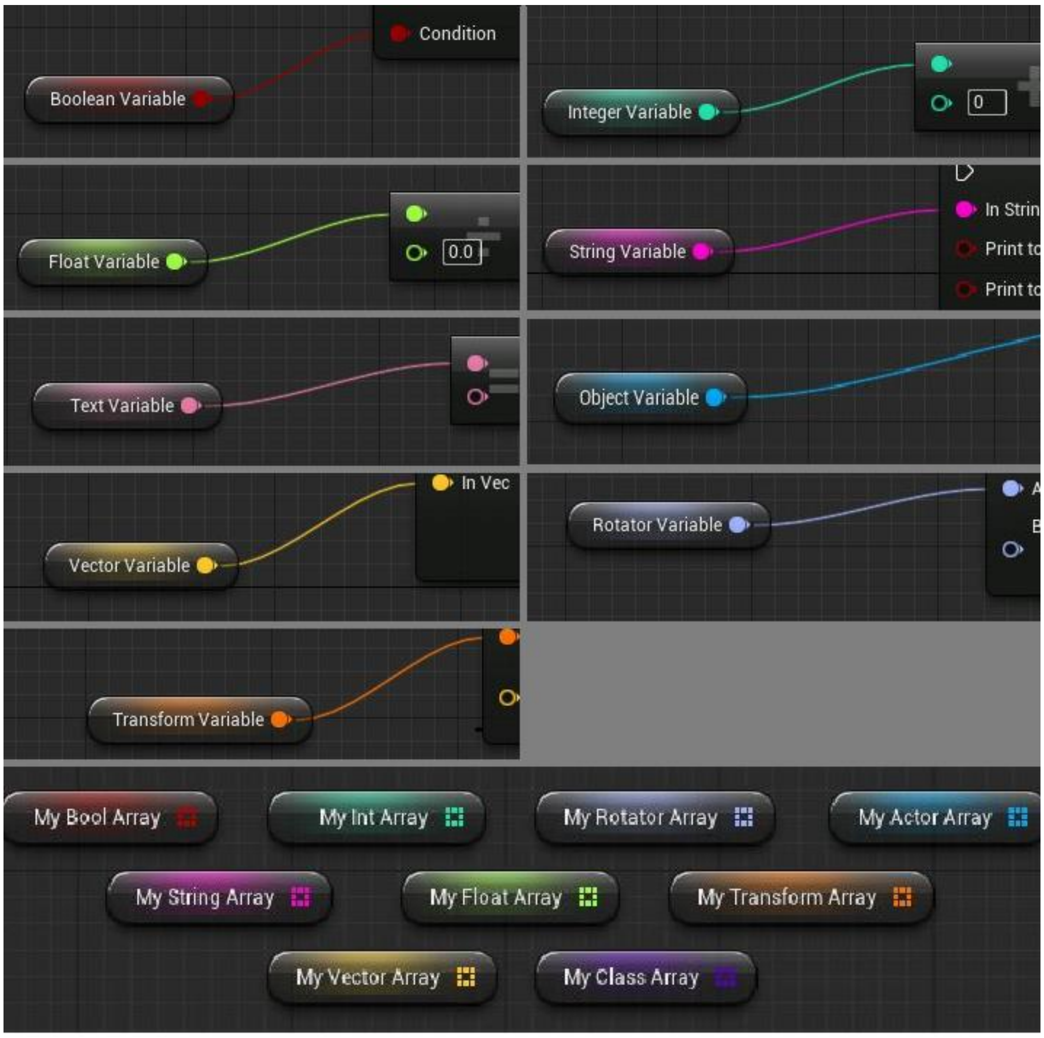


Figure 5. Basic variables used in UE blueprint visual scripting system.

Unlike in conventional imperative programming languages, the sequence where a variable is introduced is taken care of by the compiler. In general VPL's are fairly type safe, as the function input generally only accepts specific type of variable. Some visual programming environments allow for both static and dynamic type systems, example of this is Visual Basic.

As in text-oriented programming languages, many VPL's allow users to make their own functions, which speeds up workflow and often results in cleaner block diagrams. The system then makes a custom graphical element for the function, which displays the input and output nodes and the name of the custom function.

2.2 Shortcuts in programming workflow

When programmers set out to make a program, they usually start with a graphical representation of what the program should be able to accomplish and what the features are, using previously mentioned methods such as dataflow diagrams, state machines or other various visual representation methods. This is usually the most convenient way to keep track of what the end result should be. VPL's blur the line between designing and actually programming the software, which saves a lot of time in many cases.

In many VPL's testing is much simpler than it is in text-oriented programming languages, where you would need to come up with extensive set of test cases for maximum coverage. In VPL's many of the user errors are parsed out due to restrictions in how the relations are formed. This means that in many cases the dependencies between instructions are short term. If a variable is used in multiple places, it's simply branched to the function calls so it's easy to keep track of it and limit from unwanted side effects anywhere else in the program. All these factors make unit testing simpler and the basic structure of form-based VPL's are well suited for control flow analysis [5].

3 Visual programming; a natural environment

The human visual system is perfectly adapted to interpret images. Most things you do on a daily basis are guided by visual input; whether it is watching television, going to work or reading a book. Over the course of human history the brain has evolved into an excellent tool of extracting conceptual information from images in just milliseconds. According to a study conducted by Mary C. Potter and her team at MIT Brain and Cognitive Sciences department, it only takes under 100ms for the brain to identify the image, in some cases as little as 13 milliseconds [6]. VPL's take advantage of this strength and use variety of colors, cluster grouping and other techniques to increase developer's productivity by making it easier to navigate and read the source code. Refer to figure 8, where multiple methods are used to make the code easier to interpret.

There have been numerous studies examining the effectiveness of VPL's, one of which was an empirical study conducted by Rajeev K. Pandey and Margaret M. Burnett for the Oregon State University. In the study a group of 60 student programmers solve a vector and matrix manipulation tasks using Forms/3 VPL, Pascal and APL, which is a textual matrix manipulation language, it uses a variety of graphic symbols to represent functions and operators. Study results show that problem 1, in which student programmers wrote a program which appends 2 matrices, had by far better success rate with a VPL than with a more traditional programming language, Pascal. It even outperformed APL which has a multidimensional array as its main data type. In problem 2, the results between the programming languages are more in line with each other. This result was expected by the makers of the study, as the goal of the second problem was to compute Fibonacci sequence numbers, which doesn't benefit from the visual style of a VPL, however it didn't seem make a negative impact on it either [7].

Table 1. Results for problem 1, where student programmers wrote a program which appends 2 matrices.

	completely correct	**nearly correct**	**conceptually but not logically correct**	**incorrect**
Pascal	7	1	21	31
Forms/3	53	0	2	5
OSU-APL	49	3	2	6

Table 2. Results for problem 2, where student programmers wrote a program which computes the first N elements of the Fibonacci sequence.

	completely correct	**nearly correct**	**conceptually but not logically correct**	**incorrect**
Pascal	38	5	4	13
Forms/3	35	9	7	9
OSU-APL	15	3	6	36

In addition to the previously mentioned reasons VPL's are perfectly suited for small development groups where the expertise of different members of the team varies. Also the end users can take advantage of the relatively simple approach which some VPL's offer, making the communication between the client and developer more fluent.

Another study, which was conducted by Felipe Anfurritia, Ainhoa Álvarez ,Mikel Larrañaga, examines how learning conceptual skills and understanding of programming through a visual programming environment affects the student motivation. The drive behind the study was high failure and dropout rates of the modular and object oriented programming courses of the University of The Basque Country. Since many of the programming languages require learning various programming concepts before writing a single line of code, it can be taxing on the student. The makers of the study wanted to propose the use of visual programming environment, where the students could practice designing the program and understanding how it work, before focusing on the code syntax. The results of the study showed that 51% of the students found visual programming environment to positively affect their motivation and understanding of programming concepts. This percentage was even higher among the retakers of the course, where the percentage among male retakers was 75%. Surprisingly, female attendees of the course had significantly lower percentages, hovering around 40%.

In the same study, the students were asked if they would have liked to exclusively work with Eclipse instead of visual environment and the percentage was surprisingly high, 53 percent, even though most of the students found visual environment helpful to their learning and motivation. The writers of the study suspect that this is because the students perceived that text based environment to be more useful for their professional future. This shows how undervalued visual programming is in the programmer community, which I will talk about later in this thesis. However the students may feel though, the number of students who passed the exam was much higher with the new methodology, compared to the old one[8]. See figure 6 for results.

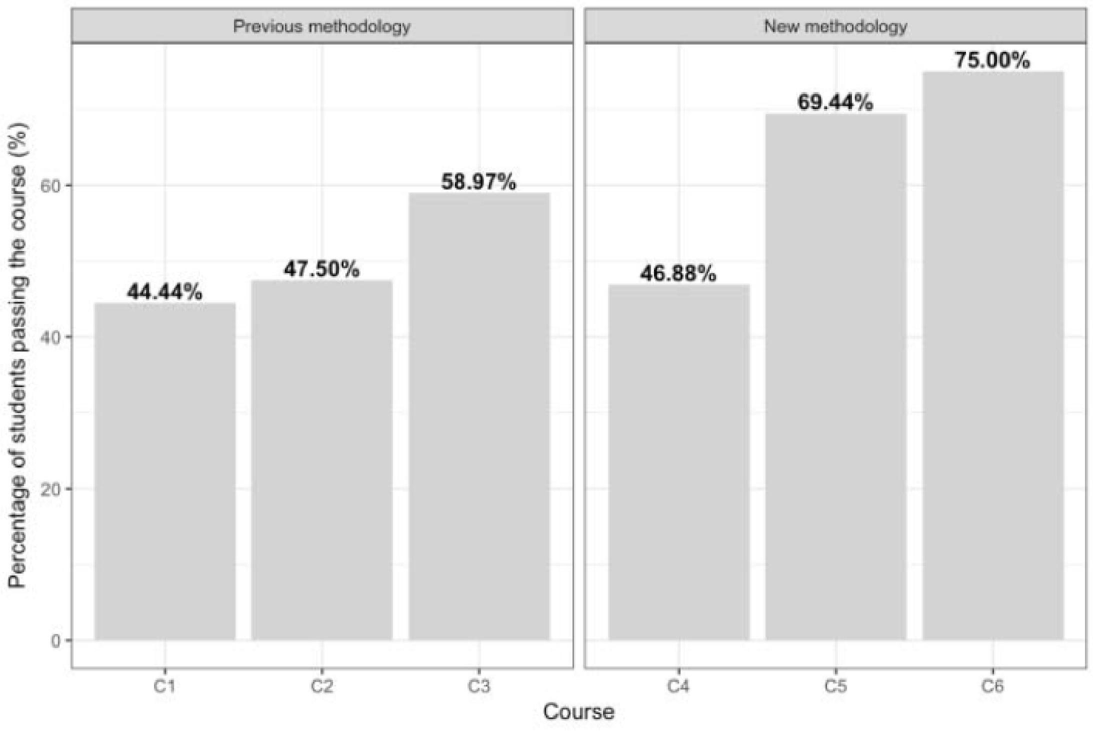
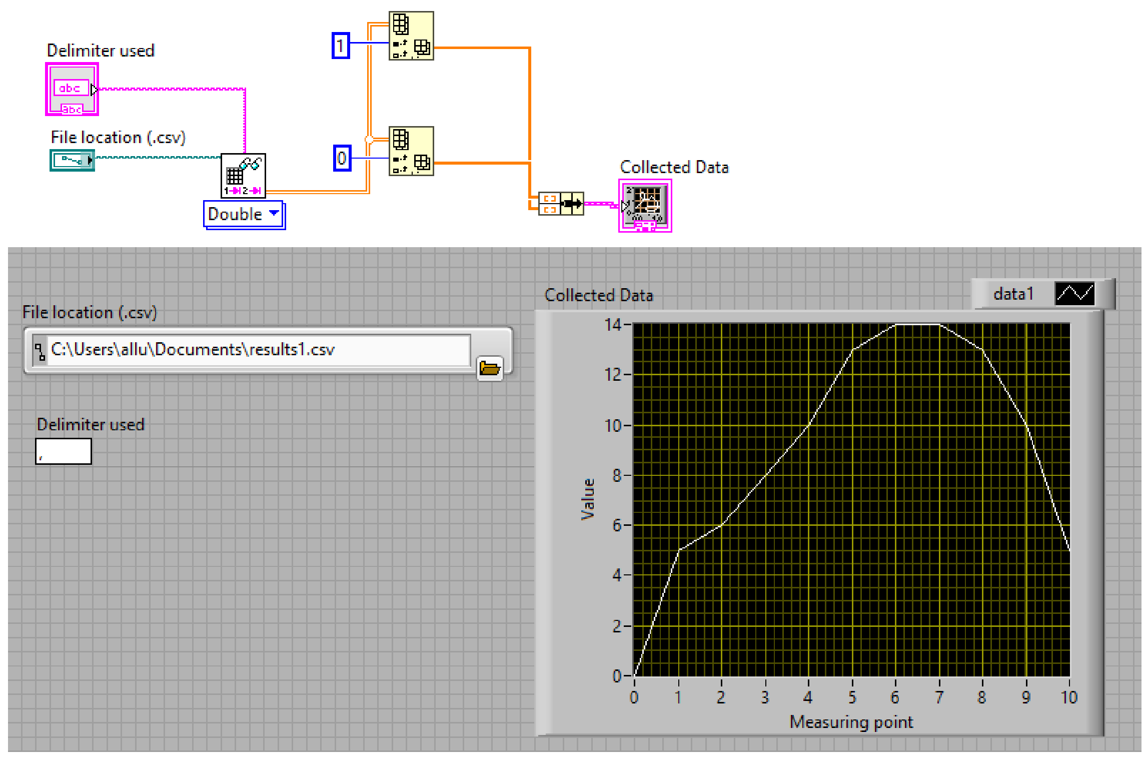


Figure 6. Percentage of students who passed the course, according to the study.

For the thesis I decided to try to make my first program in the LabVIEW VPL. Prior to this, I've only heard about it, but this is my first time actually using it. For the project, I used the evaluation version of the LabVIEW 2017. Since I didn’t have an electronic device which would collect data, I made a simple test data spreadsheet, where on row one were measuring points and on the second row was the collected data.

I started out by dragging a Read Delimited Spreadsheet-function into the block diagram. After this, I checked what kind of input it accepts and saved my spreadsheet accordingly, which was .csv in this case. There was also multiple functions in the excel tab of LabVIEW, which allow reading from xml-files directly, however I went for the more generally used Read Delimited Spreadsheet-function. Following this, I dragged a file location and string variable, which was then used for the delimiter. Read Delimited Spreadsheet-functions output was an array, which I split into two one dimensional arrays and used the cluster function which was accepted by the XY Graph-function.

The end result was a program which asks user the csv-file location for the collected data and what delimiter was used in the file. After this, it prints an xy-graph from the collected data spreadsheet. The block diagram and front panels can be seen in figure 7.



File location (.csv)
C:\Users\allu\Documents\results1.csv

Delimiter used
,

Figure 7. LabVIEW block diagram(top) and front panel(bottom).

At the top of the figure 7, you can see the node graph which defines this program. It starts from the left and moves towards the right, as do most VPLs which use the node graph system. The first two nodes are the user input, which ask what the file location is and what delimiter is used in the csv-file, which contains the measurements. After this, the function reads double-type values from the file and outputs a two-dimensional array where the first row consists of values and the second row are measuring points. Measuring points in this case could be time and values could be the electric current, temperature, humidity or other easily measurable data. After the program splits the two-dimensional array into two standard arrays and converts them from double to integer type, which is in this case the type of values that the graph drawing-function accepts. There are other various graph functions, but this is the first one which caught my eye. Double clicking on the nodes, reveal extra options. In the case of the graph node, I could have changed the range of the graph and other settings.

The use of this VPL is quite simple, with little info and experience in the VPL, I was able to make a functional program, which draws a graph from the collected data. There were a lot of functions and operations which seemed confusing at first, but each one has separate easy-to-reach documentation, which can be inspected by right clicking on the element and selecting "help". Also, I didn't use many of the programming concepts as the blocks were self-descriptive, with the exception of the two dimensional array, which was produced by the read spreadsheet function. By using it, you would need to know how the array stores the information and how the indexes work. If I were to write a similar program in say C++, it would require the knowledge of how to manipulate strings, read data from a file, usage of for or while-loops, as well as the previously mentioned array information. In addition it would require a fairly detailed knowledge of how to use an additional interface, such as graphics.h, if I wanted to draw the graph.

4 Who is VPL meant for

Throughout the years VPL’s have been maintaining their modest popularity in some specific fields, as visual programming languages tend to specialize in certain tasks, unlike traditional all-rounder languages such as C++. Especially in data analytics and robotics, VPL’s have been used for many years and still continue getting regular updates, even though initial release was over 30 years ago. This is the case with a VPL called LabVIEW, which was released back in 1986 and recently got a major update in May 2017.

4.1 VPL in game development

Unity is a cross platform game engine released in 2005. While by default programming in Unity is done in C#, it has plugins in the asset store which bring new visual scripting features, such as state machine styled animation creation, dataflow diagram programming and dozens others. Examples of this are Playmaker, Shader Forge and Terrain Composer.

Game Maker: Studio has a drag and drop system. In this system, the icons represent actions that would occur in a game, such as movement, basic image rendering and simple control structures. It’s also possible to create custom "action libraries" using the

Library Maker. Game Maker Language is the primary interpreted scripting language used in Game Maker, which is usually significantly slower than compiled languages such as C++ or Delphi.

Unreal engine has a versatile visual programming system called Blueprint. At first glance it looks like a VPL, but it is actually a visual scripting system, which can convert the graphical representation into C++ code when compiled. It is a system which takes advantage of visual programming up sides, while maintaining the complexity available in C++. If the user chooses, he may write the code directly in C++ instead of using the graphical representation. As in some other game development programming environments, UE uses data flow based graphs, with the ability to program some features such as animation transitions and AI in state machine environment [9].

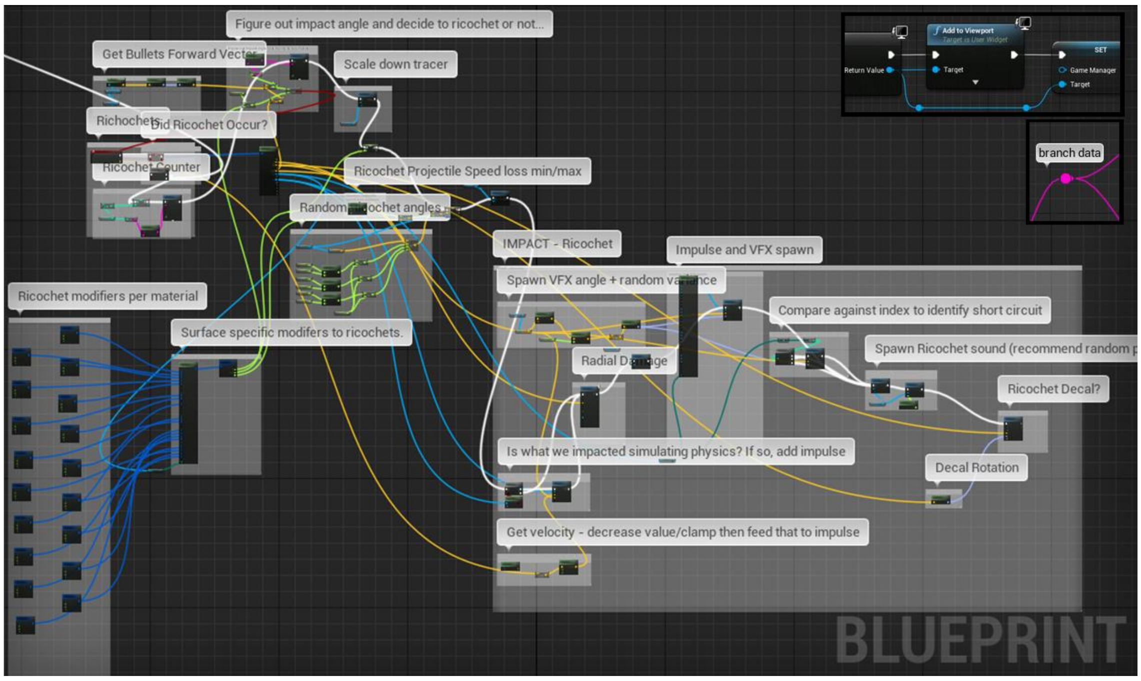


Figure out impact angle and decide to ricochet or not...
Get Bullets Forward Vector
Scale down tracer
Richochet
Did Ricochet Occur?
Ricochet Counter
Ricochet Projectile Speed loss min/max
Random Ricochet angles
IMPACT - Ricochet
Impulse and VFX spawn
Spawn VFX angle + random variance
Ricochet modifiers per material
Surface specific modifers to ricochets.
Compare against index to identify short circuit
Spawn Ricochet sound (recommend random
Radial Damage
Ricochet Decal?
Is what we impacted simulating physics? If so, add impulse
Decal Rotation
Get velocity - decrease value/clamp then feed that to impulse
Add to Viewport
Return Value
Target
SET
Game Manager
Target
branch data
BLUEPRINT

Figure 8. Bullet ricochet code made in UE blueprint system.

In figure 8, you can see the various methods of keeping the graph clean. Node grouping, commenting and variable coloring are all used to make the end result cleaner. The used may also branch and reroute the reference wires. This hasn’t been used in the graph because of the simplicity of the code, but example of branching and rerouting can be seen in the right upper corner of the image.

I have nearly two years of experience in developing with Unreal Engine and after researching many other VPL's and visual scripting languages, I'm glad to see that I made a good decision in choosing it. Many of the flaws, which I go through later in this thesis, are not present in UE Blueprint. I work in a small developer team and the simple layout of the Blueprint system allows the artists and designers to be more in sync with the programmers, which makes the team synergy and efficiency much better.

Before I started with UE, I had little previous experience with a VPL. I dabbled in the previously mentioned Game Maker: Studio, which was mainly suitable for 2D game development. When I entered UE blueprint environment, it felt familiar. It was easy to navigate, use functions and it was based on the dataflow diagram model, which I've used in the past. It took advantage of coloring, grouping and commenting features, which made the reading of the code simpler. Example of grouped and commented code can be seen in figure 8.

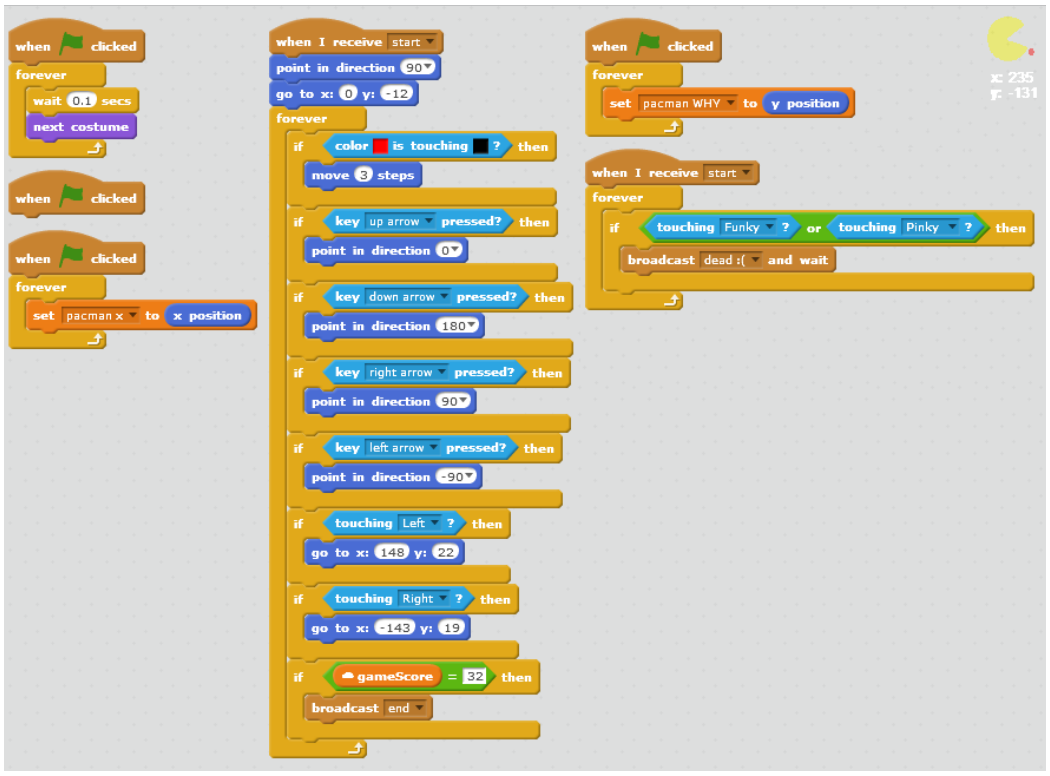
4.2 VPL in education

According to code.org inquiry in the U.S., around 90 percent of parents want their children to learn computer science. Visual programming in the form of educational games and quizzes is the perfect way for them to get started. Certain apps, such as Kodable, are suited for kids as young as five years old. Teaching the kids how to code can be very beneficial for them in the future, not because they might become a software developer when they grow up, but because it provides a better understanding of how programs work and how to work with computers in general.

Due to the low learning curve, various VPL's have been used for educational purposes. After-school programs and IT-classes have been using VPL's to develop technological fluency, logical and creative thinking. It is a useful tool for both teachers and students. Teachers can create quizzes, puzzles and games, while students can create programs which are helpful for them, such as automating some repetitive tasks.

Scratch is one of the more popular VPL's for education purposes. It was developed by Massachusetts Institute of Technology research laboratory in 2002 and it is still used to this day. It has even made its way into Harvard University introductory IT-course syllabus, with the goal of teaching the basics about statements, Boolean expressions, conditions, etc.

Scratch is an event driven VPL, it's mostly used to program interactive stories, games and animations. Its simplistic interface, which is translated to over 40 languages, is easy to understand, thus making it fitting for elementary level programming. It also features a large online community, where people share their programs and the source code. This kind of environment encourages trying different things and learning from the more advanced users of the programming language [10].



```
when ⚑ clicked
forever
    wait (0.1) secs
    next costume

when ⚑ clicked

when ⚑ clicked
forever
    set [pacman x] to (x position)

when I receive [start]
point in direction (90)
go to x: (0) y: (-12)
forever
    if <color 🟥 is touching ⬛ ?> then
        move (3) steps
    if <key [up arrow] pressed?> then
        point in direction (0)
    if <key [down arrow] pressed?> then
        point in direction (180)
    if <key [right arrow] pressed?> then
        point in direction (90)
    if <key [left arrow] pressed?> then
        point in direction (-90)
    if <touching [Left] ?> then
        go to x: (148) y: (22)
    if <touching [Right] ?> then
        go to x: (-143) y: (19)
    if <(☁ gameScore) = [32]> then
        broadcast [end]

when ⚑ clicked
forever
    set [pacman WHY] to (y position)

when I receive [start]
forever
    if <<touching [Funky] ?> or <touching [Pinky] ?>> then
        broadcast [dead :(] and wait
```

Figure 9. Pac-Man game replicated in Scratch by the community user GlitchUare.

In figure 9, you can see a simple Pac-Man game replicated in Scratch. When the user clicks on the screen the, game sets up Pac-man's position and direction. After this, it enters a loop which checks if the red dot is overlapping with the black color, which is the ground. The loop also checks for user arrow inputs and if the player is at either side of the edges of the map, in this case it will move the Pac-Man to the other side. The icon seen in the upper right corner is the player character and the small red dot next to it is the collision check color. If the red dot is not overlapping with the black color, Pac-Man has run into a wall and he will no longer be able to move in that direction. The loop also checks for user key inputs and turns the character according to the key pressed. The only two ways of getting out of the otherwise infinite loop are getting score of 32, which means you’ve collected all Pac-Dots. Another is if the player character collision is overlapping with the ghosts’ collision boxes, in which case the game is over. The position of the functions doesn’t matter as the system checks for user input separately each tick.

Scratch received 2.0 update in 2013, making the interface feel less dated and allowing users to directly edit and remix projects posted on the scratch community site, for this to work Scratch was rewritten in Adobe Flash. Another, 3.0 update is set to be released sometime in 2018. According to the Scratch website, it will feature a complete redesign and reimplementation of Scratch, as well as rewrite in HTML5.

Due to the success of the Scratch programming environment, google has decided to collaborate with the Scratch team to bring the next generation of programming blocks for kids, called Scratch Blocks. The new Scratch Blocks UI resembles the one seen in Scratch, but it is based on Googles Blockly visual code editor and they decided to add horizontal orientation, in addition to the vertical which was used in Scratch. The goal is to modernize the environment to be more suited mobile devices and make it open source, so the blocks could be easily integrated in various apps, games and other products outside the Scratch Blocks environment. The collaboration was announced on May 17th 2016, but the final release date has not been set yet. It is also uncertain if the Scratch Blocks and Scratch 3.0 projects will merge at some point in their development.

4.3 VS in media creation

When people think about any kind of programming, media creation isn’t the first thing that comes to mind. The influence of VPL’s has spread over the years and many multimedia creation and design software’s take advantage of the concepts we are used seeing in visual programming. Most used is the node graph architecture where modular nodes can be connected to each other forming a graph, these kind of graphs resemble ones seen in dataflow programming languages.

In multimedia, influence of VPL’s can be found in 3D modeling, website design, animating, 2D texture generation and even music creation. In some cases the node graph system doesn’t bring anything new to the table, it’s just an alternative way of doing the same task. However in 3D modeling and 2D texture generation the node graph architecture can offer significant advantages compared to traditional methods. For example, standard raster graphics editors use stack based method of storing actions, which means that if you want to undo a change, you have to discard the top of the stack to get to the desired step. In the graph method however, you can simply remove the undesired node from the middle and connect it back to the rest of the nodes. The user can also add whole node clusters and edit them in any order he chooses. Refer to figure 10 for visualization.

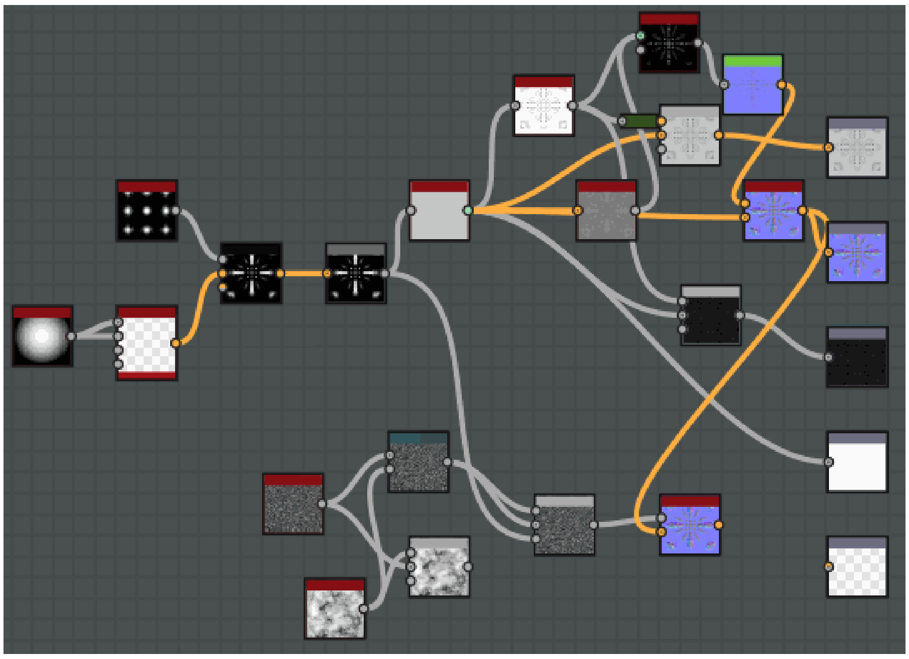


Figure 10. Node graph system used in Substance Designer.

In figure 10 you can see the node system used by the Substance designer, a texture and material authoring tool. Each node changes the end result in various ways such as blur, distort, hue shift and other ways. After each step you can see how the image changes and on the far right side you can see the output which is exported from the program. The program exports diffuse, normal, metallic, roughness and other textures which the used can define manually. These textures are then interpreted by various game engines to get the environments and items look the certain way. Alternatively, this program could be used to just export the base color and could work as a raster graphics editor program such as Photoshop.

Web page design and programming has also taken advantage of the visual programming style. Many companies offer drag and drop interfaces to clients with no or little programming knowledge. This allows users to construct professional looking web pages, without having to hire a professional to do the work. Traditionally you would need to know HTML, CSS or JavaScript to make professional looking web pages. The cost of this is flexibility and uniqueness, however that is constantly being updated, giving the users increasing amount of options and styles to choose from. Companies such as Wix, Weebly, Duda and Squarespace offer this kind of service. At the moment of writing this thesis, one of the most popular web development platforms is Wix. It uses HTML5 markup language to create the web sites, but users won't need to see a single line of HTML due to how the webpage tool is designed. Example of a Wix web page builder can be seen below, in figure 11.

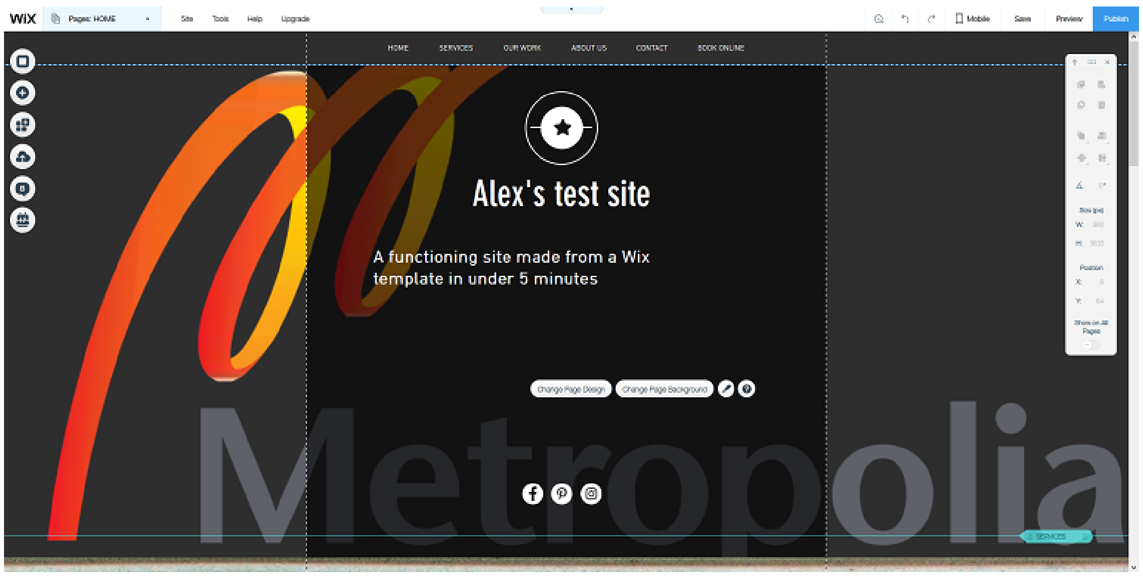


Figure 11. Wix browser web page editor

In figure 11, you can see the functioning example page made in Wix browser editor in less than five minutes, without touching a single line of HTML. It was made completely from drag-and-droppable elements available on the site, excluding the background image.

4.4 VPL in data acquisition and robotics

Previously mentioned LabVIEW visual programming language is the most popular one in this category because it supports laboratory environments well, as the name might suggest. The user can control and monitor electronic equipment via it, which can also be read directly by the program. It's also started to see more use in robotics, represented by the multiple drone and robot kit projects available for it, some of which can be found on their community site labviewmakerhub.com, where you can download the projects and use them with your own or robotics kit. Another sample of LabVIEW’s flexibility is the chess game made by the user Ouadji, seen in figure 12. It was made fully in the block editor, excluding some the graphics he used. It is a fully functional chess game including an AI with two difficulty settings, which I tried playing against and admittedly lost to.

What makes LabVIEW special is that even though it’s based on the data flow paradigm, it still allows the use of global and local variables. It also has a node called formula node, which enables part of the code to be written in a diagram using the C++ syntax structure. This can compensate for the lack of a certain function or just make a complicated mathematical operation cleaner. However, it’s worth noting that usage execution a formula node is much slower compared to the standard code. Another excellent feature is the easily available graphical representation of the program on the front panel, which can display various data easily without extensive programming. Things like graphs, indication lights, gauges and buttons can be added with a few clicks, instead of the debug style print outs frequently used in textual programming languages. All of these features mask the flaws of this data flow based language and make it a flexible and easy-to-use tool.

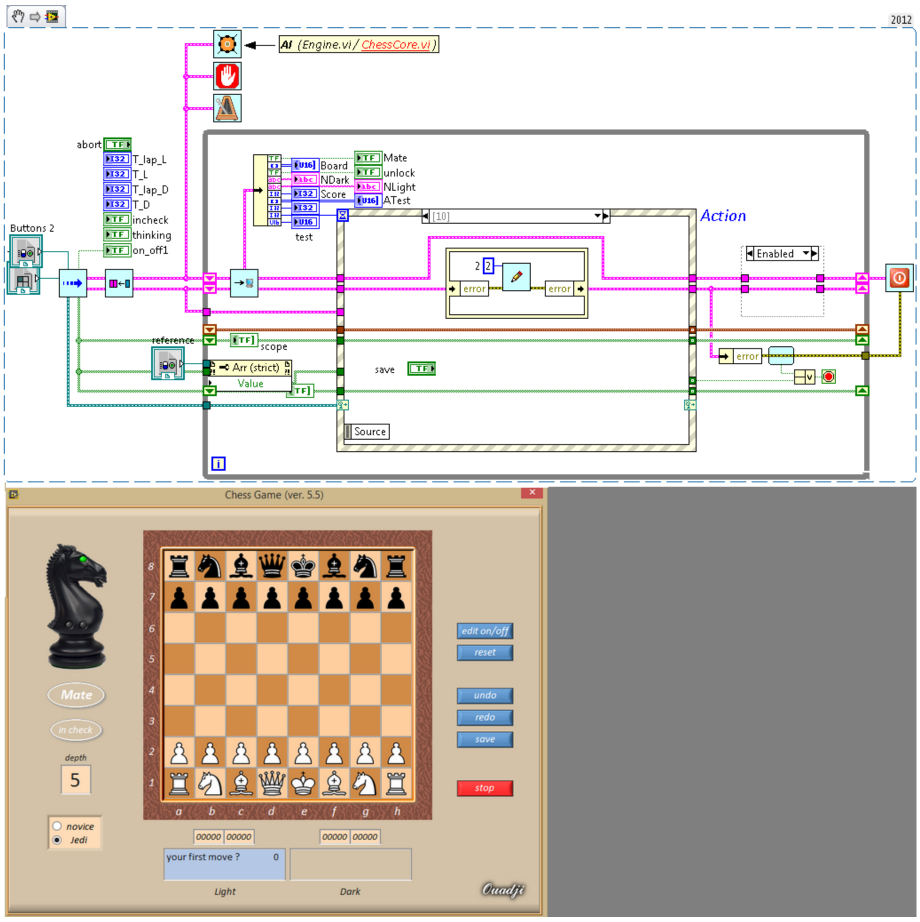


Figure 12. Fully functional chess game made in LabVIEW by the user Ouadji.

Other various VPL's have taken advantage of the increasingly popular robotics hobby kits and projects. Hobby projects using Arduino, Raspberry Pi and Lego NXT have various levels of difficulty and have many VPLs that cater to the projects. Examples of these VPLs are MindRover and NXT-G. Since the difficulty of the project and the programming capabilities of the user vary, it's important to have visual programming environments to encourage and assist people with no programming background.

5 Limitations

There are still some fundamental limitations with visual programming languages, which seemingly have no fixes. The decision on which VPL to use seems to be always linked to what kind of software you're developing and which of the limitations are acceptable for the task at hand. This is one of the biggest reasons why there is really no widely used VPL which doesn't have most of the problems described below.

The limitations can be split into 5 problem groups:

- Visual clutter and screen space
- Processing speed
- Transferability
- Conservative semantics
- Scalability

Screen space used to be a bigger problem in the past than it is now. Modern VPL's have acknowledged this issue and taken steps in the right direction to fix the issues by allowing the programmer to create group functions and reroute the nodes, making the overall result cleaner, even when the code is otherwise complex. Example of this can be seen in figure 8. Though this can be a time consuming process as VPL's generally don't have automatic refactoring feature, but the same can be said about text based programming languages, where a programmer could want to make the code more readable for the manager or a colleague. Panning and zooming are also available to the user, unlike in some of the older VPL's where navigating the grid was done with arrow keys or by dragging the scroll bar. This made some VPL's feel unnecessarily sloppy and slow. Many VPL's also allow the combining operator clusters into functions, which reduces the screen space problem. Also color coding the nodes has been helpful to improve the visibility. Even with the methods mentioned above, poorly managed node graphs are daunting to look at. Example of this can be seen below in figure 13. Currently manual work is required to make them easily readable, as there is no automatic way of cleaning up the code. However, some VPL's allow the programmer to collapse the grouped nodes, which makes even more complex systems look a bit more manageable.

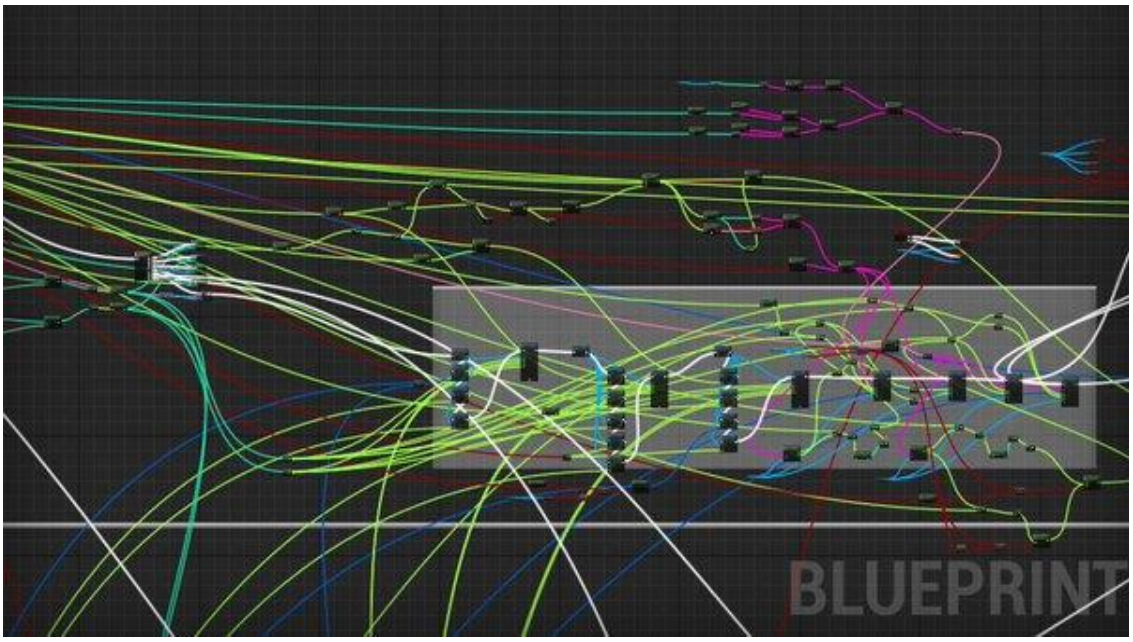


Figure 13. Poorly managed and confusing blueprint in Unreal Engine.

The rendering of images and other graphical elements will always be more taxing on the hardware than plain text. However, processing and compiling speeds have improved greatly over the years, while the VPL requirements remaining close to what they were years ago. This is still a small concern that should addressed by the programmer in some visual programming environments, where a large part of the calculations are done in real-time, such as in game development and robotics. In most present day cases though, the strain even on the most basic hardware is negligible. On the other hand, data flow based visual programming languages allow for parallel computing, which takes better advantage of the multi-core processors, as explained at the beginning of the thesis.

Conservative semantics of VPL's is at the root of many of the problems. VPL's are bound tightly to their program structure, which affects the flexibility in areas such as switch and loop structures and run-time variability. Due to the way operators and variables are expressed, sometimes it's overly complicated to complete certain tasks. Example of this is a simple for or while loop, which can be hard to achieve in some VPL's especially ones based on the data flow paradigm. This is why many visual data flow languages usually borrow elements from control-flow based languages, such is the case with LabVIEW, where it's possible to assign global and local variables, though it's considered a bad practice. Again, it's a matter of selecting the correct tool for the job and it is unlikely that there will be one silver bullet, which will fix the core of the problem.

Transferability is one of the major problems, which stands in the way of VPL's becoming popular for software development. Because of the large variety of styles in visual programming, it's nearly impossible to transfer the source code from one VPL to another. In text based languages if the programmer chooses, for whatever reason, to switch the programming language mid development. He has many tools at his disposal, which make the process manageable. Some VPL's however lack these basic features, such as search function, copying and pasting, bulk selection and replacing and many others. This makes transferring or even editing the source code problematic in many of the VPL's, slowing down workflow. One other flaw, which ties in to transferability, is the lack of version control in some visual programming language environments. Many do have in built version control, but some do not. This makes software development in lager groups very problematic, especially when combined with the previously mentioned problems. Trying to incorporate a widely used version control system such as Git, Helix VCS or Subversion, would in some cases be possible. Though this would require a lot of scripting and repackaging, which would make the process as painful as storing the saved versions manually [11].

Scalability is negatively impacted by most of the previously mentioned problems. Inside the development environment, the problem is the cluttering of the nodes and their management, as well as the execution of all available operations regardless if they are needed anywhere else, which is typical for data flow programming languages. Outside the development environment problems, such as sloppy version control, arise. Because of the visual nature of VPL's extra effort is required to make version control functional and from author's experience branch merging can be problematic and buggy. As the program grows in size, the limitations of visual programming languages become more apparent.

6 Public perception

Many programmers have strong feelings about which programming language is "the best", others have similar thoughts about VPL's. Some say it's a waste of time learning or using them, while others say they save a lot of time and couldn't do their job properly without them.

It's still a bit of a stigma in the programming community, where if you're a using a VPL you're not a proper programmer and you are not an efficient worker. I think this thinking is very outdated and people still refer to the old articles from decades ago, which stated some fundamental flaws in the VPL's of that time. However many of the problems they point out have been fixed or greatly improved in the more modern VPL's and now have many features that imperative programming languages don't have or are hard to take advantage of, such as task parallelism. VPL's biggest problem is still adapting to large-scale applications, where all the smaller problems they have are exaggerated. However, for smaller scale applications they are suited extremely well.

One of the first communities that embraced the VPL's was the scientific community. Some VPL's which were developed for data acquisition and instrument control in the 80's are still in use. Another, more recent, community has been the game developer community, where various artists and designers can be more closely in contact with the source code. This closer interaction between designers and programmers often leads to a more efficient workflow and a result that's more in line with the original vision of the designers.

7 Future of visual programming

The society is constantly moving deeper into the Information Age and I predict that in a few years, visual programming will be on par with text-oriented programming in popularity. In five to ten years, I expect most people who work with computers, will have worked with some sort of visual programming environment, be it programming, multimedia creation or education. Already it's spreading in game developer communities, it's being taught in IT-classes from a young age, it's been used by scientists and data analysts for decades and it's commonly used in robotics, which is a rapidly growing field.

There are still problems in visual programming, but it has come a long way since the 70's. Currently there are no all-rounder VPL's and the user needs to do research on which one to use. Hopefully new and improved VPL's continue to be developed, because there is more potential that can be tapped into.

8 Summary

I have been working with visual programming for nearly two years and I've come to appreciate the simplicity and effectiveness it brings to the table. However, before I started, I heard a lot of negatives about visual programming, many of which I've not encountered in my work so far. Thus, the goal of the thesis became making myself more familiar with various other VPL's and trying to figure out if the negative aspects, I read about before I started using the visual programming environment, were true and if the arguments against them were up to date.

During the research for the thesis, I found out that many of the problems, which people reference a lot, were only seen in the early stages of VPL development. Since then, many of the design flaws and inconveniences have been patched up. There are still some issues with VPL's, which can be deal breakers for users, such as the lack or poor implementation of version control in some visual programming environments. Before the research, I didn't think it was even a possibility not to have any applicable version control system, nor did I encounter any problems during my work, since the visual scripting environment I use in UE, supports SVN and Perforce-version control systems without any complicated procedures. It can also be modified to work with Git, which is one of the most popular version control systems at the moment.

Hopefully this thesis has debunked some of the commonly referenced problems and spread some awareness, which could lead to even a few more people using visual programming to develop their software. Visual programming is a powerful tool, but no single tool can complete every task, so one must weigh the pros and cons for the task at hand.

References

1 Umut A. Acar; Arthur Chargueraud & Mike Rainey. An Introduction to Parallel Computing in C++. Released on 01.03.2016, web article. Article read 14.10.2017. <https://www.cs.cmu.edu/~15210/pasl.html>

2 Arie Avnur. Finite State Machines for Real-Time Software Engineering. Released on 01.12.1990, in Computing and Control Engineering journal. Article read 18.10.2017. <https://www.researchgate.net/publication/3363248_Finite_State_Machines_for_Real-Time_Software_Engineering/>

3 Rémi Dehouck. The maturity of visual programming. Released on 29.09.2015, web article. Article read 5.10.2017. <http://www.craft.ai/blog/the-maturity-of-visual-programming/>

4 William Robert Sutherland . The On-line Graphical Specification Of Computer Procedures. Released in 1966, for Massachusetts Institute of Technology. Dept. of Electrical Engineering. Article read 02.11.2017. <https://dspace.mit.edu/handle/1721.1/13474>

5 By Gregg Rothermel; Lixin Li & Margaret Burnett. Testing Strategies for Form-Based Visual Programs. Released on 01.11.1997, for International Symposium on Software Reliability Engineering . Article read 5.11.2017. <http://cse.unl.edu/~grother/papers/issre97.pdf>

6 Mary C. Potter. Target detection at 50 or 33 ms/picture in RSVP. Released on 01.09.2011, in Journal of Vision. Article read on 30.10.2017. <https://www.researchgate.net/publication/275436083_Target_detection_at_50_or_33_mspicture_in_RSVP>

7 Rajeev K. Pandey & Margaret M. Burnett. Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study. Released on 27.08.1993, for Department of Computer Science, Oregon State University. Article read on 30.10.2017. <ftp://ftp.cs.orst.edu/pub/burnett/vl93.empirical.study.ps>

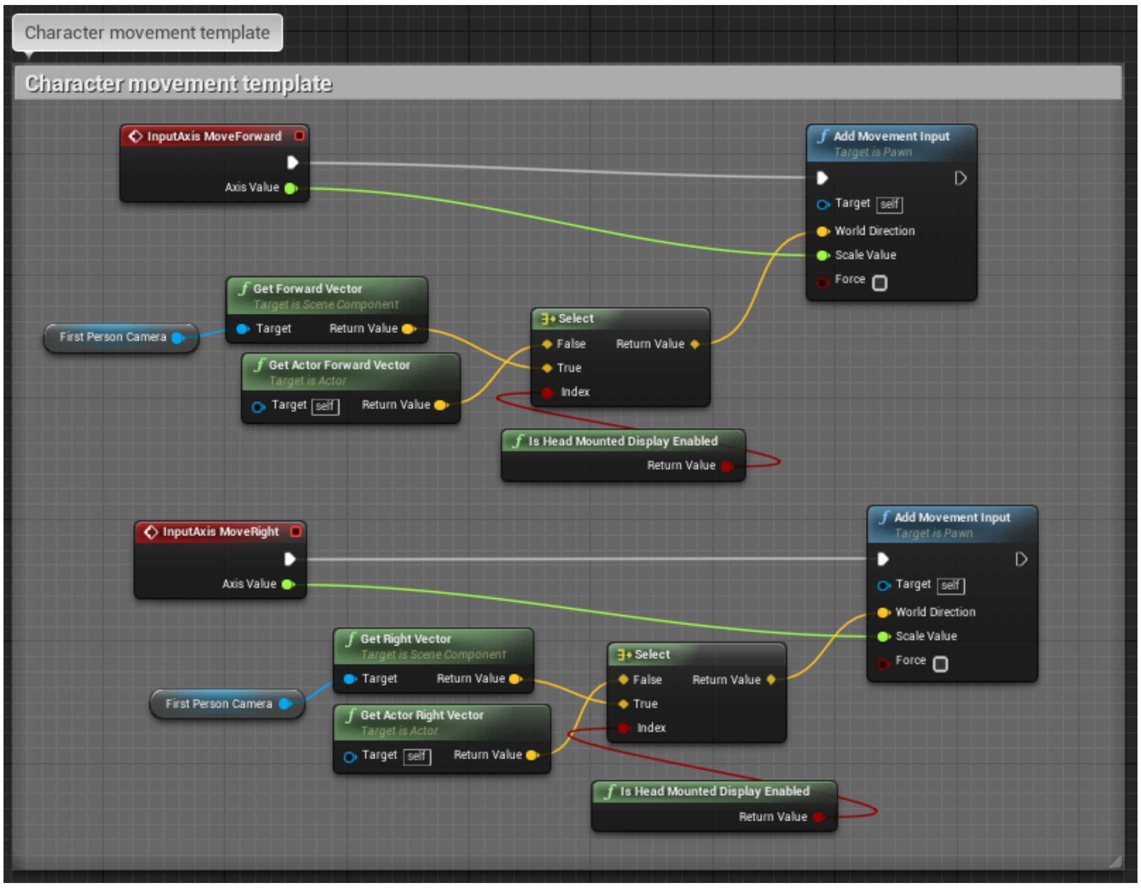
8 Felipe I. Anfurrutia; Ainhoa Álvarez; Mikel Larrañaga & Juan-Miguel López-Gil. Visual Programming Environments for Object-Oriented Programming: Acceptance and Effects on Student Motivation. Released on 07.08.2017, in Ibero-American Journal of Learning Technologies. Read on 04.12.2017. <http://ieeexplore.ieee.org/document/8003326/>

9 Epic Games, Inc, Unreal Engine 4 Documentation. Web article. Article read on 2.10.2017. <https://docs.unrealengine.com/latest/INT/>

10 Mitchel Resnick; John Maloney & Andrés Monroy-Hernandez. Scratch: Programming for All. Released on 01.11.2009, in Communications of the ACM journal. Article read on 5.11.2017.
<https://web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf>

11 Tiago Simõnes. Visual Programming Is Unbelievable… Here's Why We Don't Believe In It. Released on 25.03.2015, web article. Article read 5.10.2017.
<https://www.outsystems.com/blog/visual-programming-is-unbelievable.html>

12 Maija Marttila-Kontio. Visual data flow programming languages: challenges and opportunities. Released on 5.5.2011, Dissertation for University of Eastern Finland Dept. of Computer Science. Read on 25.03.2018
<http://epublications.uef.fi/pub/urn_isbn_978-952-61-0418-8/urn_isbn_978-952-61-0418-8.pdf>



```
#include "GeneratedCppIncludes.h"
#include "Private/NativizedAssets.h"
#include "Public/FirstPersonCharacter__pf205484891.h"
PRAGMA_DISABLE_OPTIMIZATION
#ifdef _MSC_VER
#pragma warning (push)
#pragma warning (disable : 4883)
#endif
PRAGMA_DISABLE_DEPRECATION_WARNINGS
void EmptyLinkFunctionForGeneratedCodeFirstPersonCharacter__pf205484891() {}
// Cross Module References
    NATIVIZEDASSETS_API UFunction*
Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__InpAxisEvt_MoveForward_K2Node_InputAxisEvent_181__pf();
    NATIVIZEDASSETS_API UClass*
Z_Construct_UClass_AFirstPersonCharacter_C__pf205484891();
    NATIVIZEDASSETS_API UFunction*
Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__InpAxisEvt_MoveRight_K2Node_InputAxisEvent_192__pf();
    NATIVIZEDASSETS_API UFunction*
Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__UserConstructionScript__pf();
    NATIVIZEDASSETS_API UClass*
Z_Construct_UClass_AFirstPersonCharacter_C__pf205484891_NoRegister();
    ENGINE_API UClass* Z_Construct_UClass_ACharacter();
    COREUOBJECT_API UScriptStruct* Z_Construct_UScriptStruct_FVector();
    ENGINE_API UClass* Z_Construct_UClass_UCameraComponent_NoRegister();
    ENGINE_API UClass* Z_Construct_UClass_USkeletalMeshComponent_NoRegister();
    ENGINE_API UClass* Z_Construct_UClass_USphereComponent_NoRegister();
    HEADMOUNTEDDISPLAY_API UClass*
```

```
Z_Construct_UClass_UMotionControllerComponent_NoRegister();
// End Cross Module References
    static FName
NAME_AFirstPersonCharacter_C__pf205484891_bpf__UserConstructionScript__pf =
FName(TEXT("UserConstructionScript"));
    void AFirstPersonCharac-
ter_C__pf205484891::eventbpf__UserConstructionScript__pf()
    {
        ProcessEv-
ent(FindFunctionChecked(NAME_AFirstPersonCharacter_C__pf205484891_bpf__UserCon
structionScript__pf),NULL);
    }
    void AFirstPersonCharac-
ter_C__pf205484891::StaticRegisterNativesAFirstPersonCharacter_C__pf205484891(
)
    {
        UClass* Class = AFirstPersonCharacter_C__pf205484891::StaticClass();
        static const TNameNativePtrPair<TCHAR> TCharFuncs[] = {
            { TEXT("InpAxisEvt_MoveForward_K2Node_InputAxisEvent_181"), (Na-
tive)&AFirstPersonCharacter_C__pf205484891::execbpf__InpAxisEvt_MoveForward_K2
Node_InputAxisEvent_181__pf },
            { TEXT("InpAxisEvt_MoveRight_K2Node_InputAxisEvent_192"), (Na-
tive)&AFirstPersonCharacter_C__pf205484891::execbpf__InpAxisEvt_MoveRight_K2No
de_InputAxisEvent_192__pf },
            { TEXT("UserConstructionScript"), (Na-
tive)&AFirstPersonCharacter_C__pf205484891::execbpf__UserConstructionScript__p
f },
        };
        FNativeFunctionRegistrar::RegisterFunctions(Class, TCharFuncs, AR-
RAY_COUNT(TCharFuncs));
    }
    UFunction*
Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__InpAxisEvt_Mov
eForward_K2Node_InputAxisEvent_181__pf()
    {
        struct FirstPersonCharac-
ter_C__pf205484891_eventbpf__InpAxisEvt_MoveForward_K2Node_InputAxisEvent_181_
_pf_Parms
        {
            float bpp__AxisValue__pf;
        };
        UObject* Outer =
Z_Construct_UClass_AFirstPersonCharacter_C__pf205484891();
        UFunction* ReturnFunction = stat-
ic_cast<UFunction*>(StaticFindObjectFast( UFunction::StaticClass(), Outer,
TEXT("InpAxisEvt_MoveForward_K2Node_InputAxisEvent_181") ));
        if (!ReturnFunction)
        {
            ReturnFunction = new(EC_InternalUseOnlyConstructor, Outer,
TEXT("InpAxisEvt_MoveForward_K2Node_InputAxisEvent_181"),
RF_Public|RF_Transient) UFunction(FObjectInitializer(), nullptr, (EFunction-
Flags)0x00020400, 65535,
sizeof(FirstPersonCharacter_C__pf205484891_eventbpf__InpAxisEvt_MoveForward_K2
Node_InputAxisEvent_181__pf_Parms));
            UProperty* NewProp_bpp__AxisValue__pf =
new(EC_InternalUseOnlyConstructor, ReturnFunction, TEXT("bpp__AxisValue__pf"),
RF_Public|RF_Transient) UFloatProperty(CPP_PROPERTY_BASE(bpp__AxisValue__pf,
FirstPersonCharac-
ter_C__pf205484891_eventbpf__InpAxisEvt_MoveForward_K2Node_InputAxisEvent_181_
_pf_Parms), 0x0010000000000080);
            ReturnFunction->Bind();
            ReturnFunction->StaticLink();
#if WITH_METADATA
            UMetaData* MetaData = ReturnFunction->GetOutermost()-
>GetMetaData();
```

```
            MetaData->SetValue(ReturnFunction, TEXT("ModuleRelativePath"),
TEXT("Public/FirstPersonCharacter__pf205484891.h"));
            MetaData->SetValue(ReturnFunction, TEXT("OverrideNativeName"),
TEXT("InpAxisEvt_MoveForward_K2Node_InputAxisEvent_181"));
#endif
        }
        return ReturnFunction;
    }
    UFunction*
Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__InpAxisEvt_Mov
eRight_K2Node_InputAxisEvent_192__pf()
    {
        struct FirstPersonCharac-
ter_C__pf205484891_eventbpf__InpAxisEvt_MoveRight_K2Node_InputAxisEvent_192__p
f_Parms
        {
            float bpp__AxisValue__pf;
        };
        UObject* Outer =
Z_Construct_UClass_AFirstPersonCharacter_C__pf205484891();
        UFunction* ReturnFunction = stat-
ic_cast<UFunction*>(StaticFindObjectFast( UFunction::StaticClass(), Outer,
TEXT("InpAxisEvt_MoveRight_K2Node_InputAxisEvent_192") ));
        if (!ReturnFunction)
        {
            ReturnFunction = new(EC_InternalUseOnlyConstructor, Outer,
TEXT("InpAxisEvt_MoveRight_K2Node_InputAxisEvent_192"),
RF_Public|RF_Transient) UFunction(FObjectInitializer(), nullptr, (EFunction-
Flags)0x00020400, 65535,
sizeof(FirstPersonCharacter_C__pf205484891_eventbpf__InpAxisEvt_MoveRight_K2No
de_InputAxisEvent_192__pf_Parms));
            UProperty* NewProp_bpp__AxisValue__pf =
new(EC_InternalUseOnlyConstructor, ReturnFunction, TEXT("bpp__AxisValue__pf"),
RF_Public|RF_Transient) UFloatProperty(CPP_PROPERTY_BASE(bpp__AxisValue__pf,
FirstPersonCharac-
ter_C__pf205484891_eventbpf__InpAxisEvt_MoveRight_K2Node_InputAxisEvent_192__p
f_Parms), 0x0010000000000080);
            ReturnFunction->Bind();
            ReturnFunction->StaticLink();
#if WITH_METADATA
            UMetaData* MetaData = ReturnFunction->GetOutermost()-
>GetMetaData();
            MetaData->SetValue(ReturnFunction, TEXT("ModuleRelativePath"),
TEXT("Public/FirstPersonCharacter__pf205484891.h"));
            MetaData->SetValue(ReturnFunction, TEXT("OverrideNativeName"),
TEXT("InpAxisEvt_MoveRight_K2Node_InputAxisEvent_192"));
#endif
        }
        return ReturnFunction;
    }
    UFunction*
Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__UserConstructi
onScript__pf()
    {
        UObject* Outer =
Z_Construct_UClass_AFirstPersonCharacter_C__pf205484891();
        UFunction* ReturnFunction = stat-
ic_cast<UFunction*>(StaticFindObjectFast( UFunction::StaticClass(), Outer,
TEXT("UserConstructionScript") ));
        if (!ReturnFunction)
        {
            ReturnFunction = new(EC_InternalUseOnlyConstructor, Outer,
TEXT("UserConstructionScript"), RF_Public|RF_Transient) UFunc-
tion(FObjectInitializer(), nullptr, (EFunctionFlags)0x04020C01, 65535);
            ReturnFunction->Bind();
            ReturnFunction->StaticLink();
#if WITH_METADATA
            UMetaData* MetaData = ReturnFunction->GetOutermost()-
>GetMetaData();
            MetaData->SetValue(ReturnFunction,
TEXT("BlueprintInternalUseOnly"), TEXT("true"));
            MetaData->SetValue(ReturnFunction, TEXT("Category"), TEXT(""));
            MetaData->SetValue(ReturnFunction, TEXT("CppFromBpEvent"),
TEXT(""));
            MetaData->SetValue(ReturnFunction, TEXT("DisplayName"),
TEXT("Construction Script"));
            MetaData->SetValue(ReturnFunction, TEXT("ModuleRelativePath"),
TEXT("Public/FirstPersonCharacter__pf205484891.h"));
            MetaData->SetValue(ReturnFunction, TEXT("OverrideNativeName"),
TEXT("UserConstructionScript"));
            MetaData->SetValue(ReturnFunction, TEXT("ToolTip"),
TEXT("Construction script, the place to spawn components and do other set-
up.@note Name used in CreateBlueprint function@param       Location       The
location.@param       Rotation        The rotation."));
#endif
        }
        return ReturnFunction;
    }
    UClass*
Z_Construct_UClass_AFirstPersonCharacter_C__pf205484891_NoRegister()
    {
        return AFirstPersonCharacter_C__pf205484891::StaticClass();
    }
    UClass* Z_Construct_UClass_AFirstPersonCharacter_C__pf205484891()
    {
        UPackage* OuterPackage = FindOrConstructDynamicTypePack-
age(TEXT("/Game/FirstPersonBP/Blueprints/FirstPersonCharacter"));
        UClass* OuterClass =
Cast<UClass>(StaticFindObjectFast(UClass::StaticClass(), OuterPackage,
TEXT("FirstPersonCharacter_C")));
        if (!OuterClass || !(OuterClass->ClassFlags & CLASS_Constructed))
        {
            Z_Construct_UClass_ACharacter();
            OuterClass = AFirstPersonCharacter_C__pf205484891::StaticClass();
            if (!(OuterClass->ClassFlags & CLASS_Constructed))
            {
                UObjectForceRegistration(OuterClass);
                OuterClass->ClassFlags |= (EClassFlags)0x20800080u;

                OuterClass-
>Link-
Child(Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__InpAxisE
vt_MoveForward_K2Node_InputAxisEvent_181__pf());
                OuterClass-
>Link-
Child(Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__InpAxisE
vt_MoveRight_K2Node_InputAxisEvent_192__pf());
                OuterClass-
>Link-
Child(Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__UserCons
tructionScript__pf());

                UProperty* NewProp_b0l__K2Node_Select2_Default__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("K2Node_Select2_Default"),
RF_Public|RF_Transient) UStructProper-
ty(CPP_PROPERTY_BASE(b0l__K2Node_Select2_Default__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x0010000000202000, Z_Construct_UScriptStruct_FVector());
                UProperty* NewProp_b0l__K2Node_Select_Default__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("K2Node_Select_Default"),
RF_Public|RF_Transient) UStructProper-
```

```
ty(CPP_PROPERTY_BASE(b0l__K2Node_Select_Default__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x0010000000202000, Z_Construct_UScriptStruct_FVector());
                UProperty* NewProp_b0l__K2Node_InputAxisEvent_AxisValue__pf =
new(EC_InternalUseOnlyConstructor, OuterClass,
TEXT("K2Node_InputAxisEvent_AxisValue"), RF_Public|RF_Transient) UFloatProper-
ty(CPP_PROPERTY_BASE(b0l__K2Node_InputAxisEvent_AxisValue__pf, AFirstPer-
sonCharacter_C__pf205484891), 0x0010000000202000);
                UProperty* NewProp_b0l__K2Node_InputAxisEvent_AxisValue2__pf =
new(EC_InternalUseOnlyConstructor, OuterClass,
TEXT("K2Node_InputAxisEvent_AxisValue2"), RF_Public|RF_Transient) UFloatProp-
erty(CPP_PROPERTY_BASE(b0l__K2Node_InputAxisEvent_AxisValue2__pf, AFirstPer-
sonCharacter_C__pf205484891), 0x0010000000202000);
                CPP_BOOL_PROPERTY_BITMASK_STRUCT(b0l__Temp_bool_Variable2__pf,
AFirstPersonCharacter_C__pf205484891);
                UProperty* NewProp_b0l__Temp_bool_Variable2__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("Temp_bool_Variable2"),
RF_Public|RF_Transient) UBoolProperty(FObjectInitializer(), EC_CppProperty,
CPP_BOOL_PROPERTY_OFFSET(b0l__Temp_bool_Variable2__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x0010000000202000,
CPP_BOOL_PROPERTY_BITMASK(b0l__Temp_bool_Variable2__pf, AFirstPersonCharac-
ter_C__pf205484891), sizeof(bool), true);
                CPP_BOOL_PROPERTY_BITMASK_STRUCT(b0l__Temp_bool_Variable__pf,
AFirstPersonCharacter_C__pf205484891);
                UProperty* NewProp_b0l__Temp_bool_Variable__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("Temp_bool_Variable"),
RF_Public|RF_Transient) UBoolProperty(FObjectInitializer(), EC_CppProperty,
CPP_BOOL_PROPERTY_OFFSET(b0l__Temp_bool_Variable__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x0010000000202000,
CPP_BOOL_PROPERTY_BITMASK(b0l__Temp_bool_Variable__pf, AFirstPersonCharac-
ter_C__pf205484891), sizeof(bool), true);

CPP_BOOL_PROPERTY_BITMASK_STRUCT(bpv__UsingMotionControllersx__pfzy, AF-
irstPersonCharacter_C__pf205484891);
                UProperty* NewProp_bpv__UsingMotionControllersx__pfzy =
new(EC_InternalUseOnlyConstructor, OuterClass,
TEXT("UsingMotionControllers?"), RF_Public|RF_Transient) UBoolProper-
ty(FObjectInitializer(), EC_CppProperty,
CPP_BOOL_PROPERTY_OFFSET(bpv__UsingMotionControllersx__pfzy, AFirstPersonChar-
acter_C__pf205484891), 0x0010000000010005,
CPP_BOOL_PROPERTY_BITMASK(bpv__UsingMotionControllersx__pfzy, AFirstPer-
sonCharacter_C__pf205484891), sizeof(bool), true);
                UProperty* NewProp_bpv__BaseLookUpRate__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("BaseLookUpRate"),
RF_Public|RF_Transient) UFloatProper-
ty(CPP_PROPERTY_BASE(bpv__BaseLookUpRate__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x0010000000010005);
                UProperty* NewProp_bpv__BaseTurnRate__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("BaseTurnRate"),
RF_Public|RF_Transient) UFloatProper-
ty(CPP_PROPERTY_BASE(bpv__BaseTurnRate__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x0010000000010005);
                UProperty* NewProp_bpv__GunOffset__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("GunOffset"),
RF_Public|RF_Transient) UStructProperty(CPP_PROPERTY_BASE(bpv__GunOffset__pf,
AFirstPersonCharacter_C__pf205484891), 0x0010000000010005,
Z_Construct_UScriptStruct_FVector());
                UProperty* NewProp_bpv__FirstPersonCamera__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("FirstPersonCamera"),
RF_Public|RF_Transient) UObjectProper-
ty(CPP_PROPERTY_BASE(bpv__FirstPersonCamera__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x001000040008000c,
Z_Construct_UClass_UCameraComponent_NoRegister());
                UProperty* NewProp_bpv__Mesh2P__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("Mesh2P"),
RF_Public|RF_Transient) UObjectProperty(CPP_PROPERTY_BASE(bpv__Mesh2P__pf,
AFirstPersonCharacter_C__pf205484891), 0x001000040008000c,
Z_Construct_UClass_USkeletalMeshComponent_NoRegister());
                UProperty* NewProp_bpv__FP_Gun__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("FP_Gun"),
RF_Public|RF_Transient) UObjectProperty(CPP_PROPERTY_BASE(bpv__FP_Gun__pf,
AFirstPersonCharacter_C__pf205484891), 0x001000040008000c,
Z_Construct_UClass_USkeletalMeshComponent_NoRegister());
                UProperty* NewProp_bpv__Sphere__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("Sphere"),
RF_Public|RF_Transient) UObjectProperty(CPP_PROPERTY_BASE(bpv__Sphere__pf,
AFirstPersonCharacter_C__pf205484891), 0x001000040008000c,
Z_Construct_UClass_USphereComponent_NoRegister());
                UProperty* NewProp_bpv__L_MotionController__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("L_MotionController"),
RF_Public|RF_Transient) UObjectProper-
ty(CPP_PROPERTY_BASE(bpv__L_MotionController__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x001000040008000c,
Z_Construct_UClass_UMotionControllerComponent_NoRegister());
                UProperty* NewProp_bpv__R_MotionController__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("R_MotionController"),
RF_Public|RF_Transient) UObjectProper-
ty(CPP_PROPERTY_BASE(bpv__R_MotionController__pf, AFirstPersonCharac-
ter_C__pf205484891), 0x001000040008000c,
Z_Construct_UClass_UMotionControllerComponent_NoRegister());
                UProperty* NewProp_bpv__VR_Marker__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("VR_Marker"),
RF_Public|RF_Transient) UObjectProperty(CPP_PROPERTY_BASE(bpv__VR_Marker__pf,
AFirstPersonCharacter_C__pf205484891), 0x001000040008000c,
Z_Construct_UClass_USphereComponent_NoRegister());
                UProperty* NewProp_bpv__VR_Gun__pf =
new(EC_InternalUseOnlyConstructor, OuterClass, TEXT("VR_Gun"),
RF_Public|RF_Transient) UObjectProperty(CPP_PROPERTY_BASE(bpv__VR_Gun__pf,
AFirstPersonCharacter_C__pf205484891), 0x001000040008000c,
Z_Construct_UClass_USkeletalMeshComponent_NoRegister());
                OuterClass-
>AddFunctionToFunctionMapWithOverridden-
Name(Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__InpAxisEv
t_MoveForward_K2Node_InputAxisEvent_181__pf(),
TEXT("InpAxisEvt_MoveForward_K2Node_InputAxisEvent_181")); // 3342259331
                OuterClass-
>AddFunctionToFunctionMapWithOverridden-
Name(Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__InpAxisEv
t_MoveRight_K2Node_InputAxisEvent_192__pf(),
TEXT("InpAxisEvt_MoveRight_K2Node_InputAxisEvent_192")); // 3811268844
                OuterClass-
>AddFunctionToFunctionMapWithOverridden-
Name(Z_Construct_UFunction_AFirstPersonCharacter_C__pf205484891_bpf__UserConst
ructionScript__pf(), TEXT("UserConstructionScript")); // 2692304607
                OuterClass->ClassConfigName = FName(TEXT("Game"));
                static TCppClassTypeIn-
fo<TCppClassTypeTraits<AFirstPersonCharacter_C__pf205484891> > Stat-
icCppClassTypeInfo;
                OuterClass->SetCppTypeInfo(&StaticCppClassTypeInfo);
                OuterClass->StaticLink();
#if WITH_METADATA {...}

#endif
            }
        }
        check(OuterClass->GetClass());
        return OuterClass;
    }
    IMPLEMENT_DYNAMIC_CLASS(AFirstPersonCharacter_C__pf205484891,
TEXT("FirstPersonCharacter_C"), 1314386128);
    static FCompiledInDefer
```

```
Z_CompiledInDefer_UClass_AFirstPersonCharacter_C__pf205484891(Z_Construct_UCla
ss_AFirstPersonCharacter_C__pf205484891,
&AFirstPersonCharacter_C__pf205484891::StaticClass,
TEXT("/Game/FirstPersonBP/Blueprints/FirstPersonCharacter"),
TEXT("FirstPersonCharacter_C"), true,
TEXT("/Game/FirstPersonBP/Blueprints/FirstPersonCharacter"),
TEXT("/Game/FirstPersonBP/Blueprints/FirstPersonCharacter.FirstPersonCharacter
_C"), nullptr);
    DEFINE_VTABLE_PTR_HELPER_CTOR(AFirstPersonCharacter_C__pf205484891);
PRAGMA_ENABLE_DEPRECATION_WARNINGS
#ifdef _MSC_VER
#pragma warning (pop)
#endif
PRAGMA_ENABLE_OPTIMIZATION
```