Recommender systems in visual programming environments

Phil de Groot
NHTV University of Applied Sciences
Breda
Email: phildegroot5@gmail.com

Robert Grigg
NHTV University of Applied Sciences
Breda
Email: grigg.r@ade-nhtv.nl

***Abstract*—Recommender systems help coding productivity by providing interactive suggestions during the editing process. Such systems help to keep the coders in the flow of the editing process by relieving some of the cognitive load. Little work, however, is done in integrating a recommender system in a visual programming environment. In this paper we explore implementing a recommender system to provide suggestions in the Blueprint visual programming environment found in Unreal Engine 4. We have found that using our suggested method we are able to provide an accurate suggestion in a 5-entry short-list in 73% of situations. User tests have shown that perceived productivity increased when using the implemented recommender system, however, timings taken from the same user tests showed that actual productivity decreased by 4%.**

I. Introduction

Most modern game engines include a visual programming environment (VPE) for asset creation or editing. The Blueprint editor in Unreal Engine 4 [1] and Flow Graph in CryEngine [2] allow for the creation of logic sequences and behaviors in an environment that is friendly to all professions found on a game production team. While building graphs in these VPEs the user has to switch between two main tasks: 1) translating the purpose of the graph into nodes and 2) finding the required nodes for the desired operations. Finding the correct node in lists that contain hundreds of node-types can be challenging. Through search functions this challenge can be partly reduced, but still requires the user to know the exact name to efficiently find the desired node. Partial search terms can still lead to a significant number of results the user has to browse through (searching for 'location' yields 40 results in Blueprint). These searches for nodes can take significant time and break the chain of thought while creating graphs.

In regular programming a similar problem exists. The programmer has to remember the exact name of variables and functions in order to use them. Spelling or capitalization mistakes will result in compilation errors. To solve this problem most programming environments help the user by providing an interactive set of suggestions to the user as to what they might intend to use while writing code. The purpose of this paper is to investigate the implementation of such a recommender system in a VPE.

The remainder of this paper is structured as follows. The next section further defines the problem. Section III discusses previous work done in the field of recommender systems. Section IV describes the proposed implementation. Section V discusses the test design and in section VI we present results obtained from these tests. Section VII summarizes the paper. Finally, section VIII presents some areas for further research.

II. Problem Definition

Most modern coding development environments provide the user with suggestions while writing code. Such systems have generally been accepted to increase productivity by allowing the user to insert long names with few interactions and lowering the cognitive load on the user by relieving them of the need for remembering the exact name for each variable and function [3, 4, 5, 6]. There is little work, however, done on integrating such a suggestion system to a VPE. The purpose of this paper is to explore and implement a recommender system in the Blueprint VPE of Unreal Engine 4 [1] in order to assess whether or not productivity is affected by such a system.

With this research we propose two hypotheses:

H1. Productivity will increase in a visual programming environment using a suggestion system.

H2. Context is important to provide accurate suggestions in a graph-based environment.

It has been accepted that productivity increases in a coding environment in using a suggestion system such as Intellisense [7] found in Microsoft's Visual Studio. We suspect that productivity will also increase by implementing a suggestion system in a VPE, but not for the same reasons. Productivity increases in a coding environment due to the user having to type only a partial name for a function, variable or type and let the tool complete the rest. As a secondary effect this allows the user to only remember partial names in order to use them. In a VPE very little typing is required as we observed that most of the interaction is done through the user interface of the environment. By providing suggestions for predictions of nodes that the user might require we can reduce the time required for navigating the user interface in order to find the correct node.

In this research we will integrate a recommender system in the Blueprint system found in Unreal Engine 4. We opted to go with Unreal Engine 4 as it is the latest iteration of an industry standard game engine and provides full source-code access giving us complete freedom. For the suggestions we will limit ourselves to the traditional style of suggestion

systems by providing a suggestion of a single node that might be used.

III. Previous Work

A. Text based recommender systems

There are a number of algorithms used for prediction of text and/or code. Most of the prediction structures are built for plain text, as this is the most popular use case (including coding environments and search suggestions for websites). In this section we will discuss work done in the field of prediction for text and/or code.

Virtually all modern search application employs query auto-completion to aid the user in their search effort [8]. A popular choice for query and word auto-completion is the use of a trie. The trie, also called a prefix graph, is an ordered tree data structure. Each node contained in the graph does not store the full key but only a portion of it. To reconstruct the full key for a specific node the value of the parent node must be recursively prepended to the current key value. Once this process has reached the root of the trie the full key is reconstructed.

The trie data structure allows for efficient lookup for completions given an existing string by taking the edge corresponding to the next character at each node in the current search string. If we have the string 'te' for which we will use the example trie found in Figure 1, the suggestion generation process would work as followed: at the first node the 't' edge would be taken and at the second node the 'e' branch is be taken landing us at the 'te' node. This node has the 'tea', 'ted' and 'ten' leaf nodes which can then be presented to the user as completion suggestions. In a real-world environment the trie will contain a large number of leaf nodes, for instance all the words in the English language. Many variations exist on this base trie model to increase effectiveness in certain situations. An example is the completion trie [8] which encodes scores in the trie in order to improve the quality of suggestions. Each node contains the highest score found in the child nodes, which can then be used by an A* algorithm to find the highest ranking suggestions given a prefix.

The n-gram is a statistical model for language and was originally introduced by Shannon in 1948 [9]. N-grams have recently also been applied to code completion [10]. N-grams are able to model sequences of symbols in order to create patterns from which predictions can be made. A sequence in an n-gram is always of n-length. Building a tri-gram (3-gram) of the string "abcde" would result in 3 tri-grams: "abc", "bcd" and "cde". Creating an n-gram from large sample sets (such as code bases and/or dictionaries) allows for prediction of probabilities for the next symbol given an input of n-1 length. From the resulting n-grams that contain our n-1 length prefix symbol set all the suggestions (the last symbol in the n-gram) are collected. The probabilities for each suggestion are calculated by normalizing the number of uses such that the sum of all the resulting suggestion probabilities equals to one. This model can be extended to use tokens or sets of tokens for each symbol in order to provide suggestions on a function or variable level in code as suggested by Ohara et al. [10].

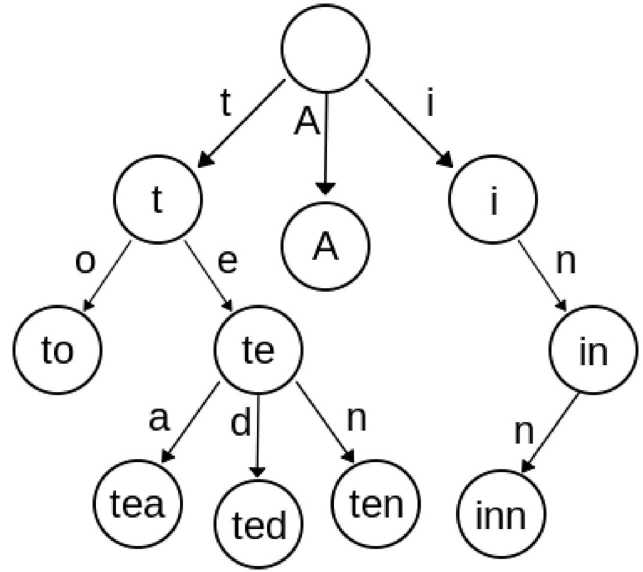


Fig. 1. Example trie.

The recently introduced cache language model extends the n-gram model by capturing local regularities in a small local cache [11]. This model works by maintaining two n-gram sets: a global (static) set built from all the available data and a local (dynamic) set constructed from local data. The local set is generated from local data around the location the user is editing. This can include the entire file the user is currently editing, or only a small portion of L lines around the cursor location. As described by Tu et al. [11] source-code is a highly local language, meaning that a significant portion of the encountered tokens are only found in a single file. According to tests performed by Tu et al. 25% of tokens in the Java corpus and 42% in the Python corpus were only found in a single file. Incorporating the suggestions found from the local set as well as the suggestion acquired from the static set significantly improved the quality of suggestions [6].

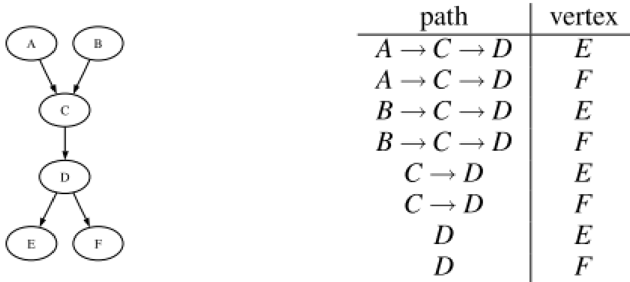
Most prediction algorithms work by suggesting a completion of a word or name. There are, however, certain solutions allow the user to work on a higher level by providing suggestions for entire snippets of code. GraPacc [12] is such a completion algorithm. GraPacc tries to provide the user with suggestions of entire snippets of code, allowing the user to focus on what they want to do rather than how they want to do it. GraPacc generates predictions via the use of groums [13]. A groum is an acyclic graph-based representation of a pattern (i.e. a snippet of code where function calls and variables are represented as nodes). Groums are created from the existing context and used to query the suggestion database for a partial match. A set of different groums is returned from the database and if one is accepted by the user the missing parts provided by the accepted groum are tailored to the existing code for seamless integration, providing a snippet based suggestion

system.

B. Graph-based recommender systems

With VPEs becoming more popular in recent iterations of big game engines (examples include Blueprint in Unreal Engine 4 [1] and Flow Graph in CryEngine [2]) efficient workflow in these graph-based environments is becoming more important. Little work, however, is available regarding exploration of recommendation system in graph-based environments [14].

VisComplete [15] is a system that was built to generate and provide suggestions for the visualization of data within VisTrails [16]. Within this work Koop et al. describe a path-based approach for mining data-graphs as well as a method for generating the predictions. The method described in this work has similarities to an n-gram approach as they both generate subsets of data. The path-based approach, however, keeps track of two pieces of data: the anchor vertex and the context path. The anchor vertex is the prediction for each set of nodes defined in the context path. The context path contains a set of nodes that are connected to this node. The nodes contained within the context path all lead in one direction, either forward (input -> output) or backwards (output -> input). A second difference from the n-gram approach is that the path does not have a fixed length, only a maximum length. An example of output data created by this algorithm can be seen in Figure 2.



path	vertex
$A \to C \to D$	E
$A \to C \to D$	F
$B \to C \to D$	E
$B \to C \to D$	F
$C \to D$	E
$C \to D$	F
D	E
D	F

Fig. 2. Forward path suggestions created for vertex D.

Bruch et al. [4] discussed that properly taking the context in account for finding suggestions will result in more accurate suggestions. By tailoring the suggestions to the specific context in which they are used will lead to more precise suggestions, while leaving the context out of the equation will result in more general suggestions. The contextual information in a graph-based environment, such as Blueprint, is drastically different from a textual or coding environment. In a strong-typed object-oriented programming language like C++ we have access to contextual information such as the object operated upon and the function containing the current code. With these pieces of contextual information a set of suggestions that do not apply to our object can already be filtered out. In a VPE there is less contextual information that is readily accessible. In a VPE, such as Blueprint, the contextual information is restricted to the data-type the next node must have and the node it has to connect to provided the user tries to create a new node from dragging a pin on an existing node.

IV. Implementation

A. Proposed prediction structure

For our proposed implementation an optimistic completion algorithm has been selected in an attempt to provide entries that have the highest probability to be correct, while minimizing the number of entries returned. This approach has been selected because we aim to induce less cognitive load on the user than a pessimistic completion algorithm which provides all possible suggestions that the algorithm is able to find [5]. In addition to the lower cognitive load the default node creation menu in the Blueprint environment already provides a pessimistic completion tool that shows all possible nodes the user can spawn.

For the implementation a prediction algorithm inspired by n-grams and the path-based approach discussed by and Koop et al. will be used [9, 15]. The inspiration from the node-based approach was taken as this approach has proven to work effectively in a graph-based environment. The implementation will contain three pieces of data: the context path, anchor vertex and prediction vertex. This data-structure contains an extra prediction vertex field when compared to the path-based structure outlined by Koop et al. and will be added for more clarity and increased ease of indexing of the data-structure. In the new data layout the anchor vertex will describe the node-type for which the suggestion applies to. The context path, conversely, will describe the previous connected nodes to the anchor vertex and the prediction vertex will describe a suggested node-type given a particular anchor vertex and context path.

B. Selected data gathering method

Creating entries for the suggestion database has been identified to be performed in two situations. 1) The user connects two nodes or 2) the user triggers a rebuild of the suggestion cache. In the first situation only the two nodes which were connected will need to be parsed for new entries. This aims for the recommender system to gather data on node use during user editing. The rebuild situation will clear the complete suggestion database and will build new suggestions using all available nodes found in the Blueprints contained in the currently open project.

For each parsed node connected nodes will be explored in a recursive manner to create the connection paths. This can be done in both the forward and backwards directions separately. If the node connection path is large enough (i.e. it contains more than two entries) a suggestion entry will be created. The examined node will be used for the prediction vertex, the first node of the connection path is to be used as the anchor vertex and the remainder of the connection list will function as the context path. Table I shows an example of suggestions that would be created when parsing node E and F for the graph found in Figure 3. The generated suggestion entries will then be added to one of the lists found in the database depending on the direction the connected nodes were parsed in (forward or backwards). It is intended that separate

lists for the two directions will be maintained to prevent the suggestions becoming mixed between directions.

TABLE I
FORWARD SUGGESTION TABLE FOR FIGURE 3.

Context Path	Anchor Vertex	Suggestion Vertex
A -> C	D	E
B -> C	D	E
C	D	E
A -> C	D	F
B -> C	D	F
C	D	F

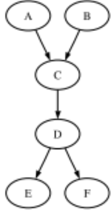


Fig. 3. Suggestion graph for Table I.

C. Proposed suggestion generation algorithm

Suggestion generation happens when the user triggers a query to the system. This is only done when the user tries to create a new node in the graph by dragging a new connection from a pin on an existing node. To generate appropriate suggestions for a particular node the system goes through several steps in order to find the most relevant suggestions. Each step further refines the results obtained from the previous step.

Step 1: Create context information: in the first step context information is constructed. A portion of the information is received as input to the query. This input includes the graph, node and pin for which suggestions are queried. Other information, such as the context paths, is created within this step. All available context paths are reconstructed through recursively following connections on the connected nodes through to a certain depth in the reverse direction of the suggestion. If a suggestion is queried for a forward direction (output -> input) the context paths are created by exploring all connections in the backwards direction (output -> input). All constructed context paths (there can be multiple) are stored within the context information structure.

Step 2: Find all suggestions for the current node in the database: the node-type that is described in the context data is used to query the suggestion database. The list that is returned contains all the suggestions that have the specified node type as their anchor vertex and are suitable for the queried direction.

Step 3: Remove incompatible suggestions: all incompatible entries are removed from the list of suggestions based on data-types. Each pin has a data-type associated with it, for example integer, float or string. Connections between two pins can only be created with the same data-types. All entries in the list that contain a prediction vertex which does not have a pin that can be connected to the pin specified in context data are removed.

Step 4: Calculate context similarities: the similarity between the suggestion context path and all available node context paths are calculated. For each context path in the context information a suggestion item is emitted with the context similarity and the prediction vertex node-type.

Step 5: Combine suggestions: all suggestions that were emitted in the previous step are processed and all suggestions having the same prediction vertex are collapsed into one. This ensures that every entry in the output list contains an unique suggested node-type. When two entries are collapsed into one the highest context similarity score is selected and used for the collapsed node. The number of uses from both entries are added together and used for the new collapsed entry.

Step 6: Select top-n suggestions: the remaining entries are sorted based on the context similarity score and number of uses. From this sorted list the top-n highest ranking entries are returned and displayed to the user.

V. TEST DESIGN

A. User tests

To test the first hypothesis defined in this document (H1. see Section II) a user test is designed. This user test was designed to provide results on how much productivity is affected by the implemented recommender system. For this test participants will be asked to solve a predefined task using the Blueprint system. The participants will be required to solve the specified task three times. The first time will be performed as a trial run to familiarize the participant with the task and allow them to find a solution that works. The second and third runs they will be asked to solve the same task again. One of the two runs will have the suggestion extension enabled and the other run will have the suggestion system disabled. Each iteration of solving the same problem should become easier for the participant due to more familiarity of the task at hand. Which of the two runs will have the extension enabled is randomized for each participant in an attempt to offset this learning effect as much as possible. The second and third runs will be recorded via screen capture to provide timings on how long participants take to solve the specified problem.

After the participants complete their three test run cycles they will complete a small survey. This survey is used to gather subjective results on how they felt the system performs. The

TABLE II
USER TEST RESULTS. (TIM SHOWS TIME IN MENU)

Participant	First Run Enabled	Without	With	Difference	%-Difference	Without TIM	With TIM	Difference	%-Difference
1	Yes	1:25	1:38	0:13	13%	0:24	0:27	0:03	11%
2	No	6:52	6:35	-0:17	-4%	1:32	2:27	0:55	37%
3	No	6:55	5:50	-1:05	-15%	1:32	1:30	-0:02	-2%
4	Yes	1:58	2:30	0:32	21%	0:37	0:30	-0:07	-19%
5	No	3:20	3:02	-0:18	-9%	0:39	0:57	0:18	31%
6	Yes	10:44	11:45	1:01	9%	2:58	2:45	-0:13	-7%
7	No	4:12	3:42	-0:30	-12%	1:22	1:07	-0:15	-18%

survey will also provide feedback on the implemented system to identify potential usability issues.

B. Context importance test

To test the second hypothesis defined in this document (H2. see Section II) an automated test will be set up. We can objectively answer whether or not the quality of the suggestions is impacted positively, negatively or not at all from taking the context into account while making suggestions by performing two automated tests and comparing their results. One of these tests will have the context calculations disabled. According to Bruch et al. [17] an automated test gives us more objective results than manual tests and allows for larger test sets. Furthermore, they suggest to use the 10-fold cross-validation technique to obtain the results as this method is proven to give accurate results [18].

In a 10-fold cross-validation the available data is split up into 10 different equal-sized sets, called the folds. Nine of these folds are used as training data for the system. The final fold is treated as testing data and is used as validation to assess whether or not suggestions are valid. This process is repeated k times, where k is equal to number of folds specified, in the case of a 10-fold cross validation k = 10. With each iteration the fold selected as the testing data is changed in such a way that after finishing all iterations each fold is used as testing data once.

Bruch et al. define two measures of the performance for a recommender system: recall and precision [17]. Recall measures the completeness of the recommendations expressed as the number of valid suggestions made divided by the number of suggestions the system should have made. Precision measures the relevance of all suggestions which is measured as the total number of suggestions made divided by the number of useful suggestions made.

Due to the design of the recommender system described within this paper measuring recall would not provide interesting results. The default implementation provided by UE4 should already provide us with 100% recall as it lists all the available nodes for a specified connection type. Precision, however, is an important metric to test for. The higher the precision we can achieve the more effective the system should be as more of the suggestions are potentially correct. If we were to achieve a 100% precision, i.e. every time the system is queried for a suggestion it is able to produce the expected node-type in the suggested short-list, we can maximize efficiency due to the search process being confined to this short-list of suggestions. We will measure precision in a different manner in the performed tests, however, where a query will pass the precision measurement if the expected node is contained within the returned suggestion short-list presented to the user.

VI. RESULTS

A. User tests

The full timings for the user tests can be found in Table II. First Run Enabled specifies whether or not the first of the two runs had the suggestion system enabled. The Without and With columns show the times it took the participant to complete the task without the suggestion system enabled and with the system enabled. Without TIM and With TIM show how much of the time the participants spent in the node spawning menu.

All participants were able to solve the problem in less time the third run than in their second run. On average participants were 12% faster in their third run. Participants who had the system enabled the second run were on average 14% faster in their third run, while participants who did not have the system enabled in their second run were only 10% faster in their third run. This indicates that the users were slightly slower (4%) to complete their task when the system was enabled, as on average the participants took more time on their second run than the third run when the system was active.

A situation we saw in the tests was where the participant inspected all the suggestions presented to them in the suggestion short-list before starting a manual search for their required node-type once they found that it was not contained in the short-list. In this situation the search time is prolonged by the addition of the search through the suggestion short-list. On average the participants spent 5% more time in menus (see Table II) when the suggestion system was enabled.

Another situation that occurred during the tests was that on several occasions the participants would ignore the suggestions provided by the system and immediately start a manual search. This could be caused by the way the participants are used to working with Blueprint, for instance by first creating nodes and connecting them afterwards. Such a workflow renders the implemented suggester system useless as it cannot generate suggestions in this manner. Changing the way someone interacts with a system they are already familiar with requires a change over a longer time, for which this test was too short.

B. User test survey

The user test survey contained 4 questions that participants could rate from 1 to 5, where 1 is to strongly disagree with the statement and 5 is to strongly agree. The full results of the survey can be found in Table III.

(Q1). Did the system provide relevant suggestions? This question gives an indication of the perceived quality of the suggestions made by the system. The participants on average rated this statement a 4.1 thus strongly agreeing with this statement. From this we can conclude that the suggestions presented by the system on average are relevant to what the user was intending to do.

(Q2). Did the system rank suggestions correctly by relevance? This question gives an indication of how well resulting suggestions are sorted when presented to the user. The participants on average rated this statement a 3.1 meaning they feel slightly more positive than neutral. Most participants mentioned they did not noticed anything particular about the sorting order of the system. From this we can say that the sorting order in the current system does not stand out to be excellent or terrible and is currently sufficient.

(Q3). Did the system speed up development time? This question is aimed to identify whether or not the system speeds up development time. On average the participants felt that the tool speeds up development time ranking this question with a 3.9. This is in contrast to the results we gathered from the user tests where on average the participants were slightly slower when the system was enabled.

(Q4). Is the system well integrated within the environment? If the system is not easy to use or well integrated in the environment the time potentially saved by using suggestions can be offset by the time it takes to correctly use the suggestion system. The participants were very positive about this and scored this question the highest with an average mark of 4.4. This indicated that the tool is perceived as well integrated within the environment and does not have any major problems in usability.

TABLE III
USER TEST SURVEY RESULTS

Participant	Q1	Q2	Q3	Q4
1	4	3	4	5
2	3	3	3	4
3	5	3	4	4
4	5	4	5	4
5	4	2	4	5
6	4	4	3	5
7	4	3	4	4
Avg	4.1	3.1	3.9	4.4

C. Context importance result

Two 10-fold cross validation tests were performed. The first test had the system compare context similarities and sort all the available suggestions based on the similarity of the contexts. The second test had the calculation of contexts disabled and as a result the suggestions were sorted on the number of uses found for each suggestion. In both tests the short-list included the top-5 suggestions for each query. This short-list was checked for the expected node-type. A query was marked as passed if the suggestion short-list included the expected node-type. The tests were performed on a set of Blueprints containing about 8000 nodes. The nodes were gathered from all available Blueprints, shuffled in a predictive manner and split up into 10 folds. As a result of all nodes not having the same amount of valid connections which we can perform a test on the number of tests performed between folds fluctuates slightly.

The results for the two tests can be found in Tables IV and V. The precision column in both tables shows the percentage of tests performed to pass our precision test. From these results we can see that taking the context into account contributes to significantly more accurate suggestions being made. The overall results improved by almost 17% when the context is taken into account. The R1 to R5 columns show where the correct suggestions show up in the suggestion short-list, R1 being the first entry and R5 being the last. From this we see a distinct increase in the amount of correct suggestions found in the higher ranks of the suggestion shortlist when the context calculation is enabled. An average of 10% more of the correct suggestions made by the system appear in the first rank when context calculation is enabled.

TABLE IV
CROSS VALIDATION RESULT WITH CONTEXT

Fold	#Tests	#Passed	Precision	R1	R2	R3	R4	R5
1	1546	1135	73.41%	62%	23%	9%	4%	2%
2	1652	1241	75.12%	60%	23%	10%	4%	3%
3	1594	1133	71.07%	64%	22%	7%	4%	3%
4	1592	1156	72.61%	63%	22%	9%	4%	2%
5	1656	1212	73.18%	63%	22%	8%	4%	3%
6	1571	1164	74.09%	66%	21%	8%	3%	2%
7	1638	1182	72.16%	62%	24%	8%	4%	2%
8	1634	1215	74.35%	63%	23%	8%	4%	2%
9	1630	1224	75.09%	64%	21%	9%	3%	3%
10	1644	1213	73.78%	61%	24%	10%	4%	1%
	16157	11875	73.49%	63%	22%	9%	4%	2%

TABLE V
CROSS VALIDATION RESULT WITHOUT CONTEXT

Fold	#Tests	#Passed	Precision	R1	R2	R3	R4	R5
1	1546	855	55.30%	53%	22%	12%	7%	6%
2	1652	997	50.35%	50%	22%	13%	8%	7%
3	1594	884	55.45%	55%	21%	10%	9%	5%
4	1592	864	54.27%	53%	24%	10%	8%	5%
5	1656	910	54.95%	52%	21%	12%	9%	6%
6	1571	917	58.37%	54%	21%	12%	6%	7%
7	1638	886	54.09%	50%	25%	12%	7%	6%
8	1634	937	57.34%	54%	24%	11%	7%	4%
9	1630	945	57.97%	54%	23%	10%	7%	6%
10	1644	946	57.54%	52%	21%	14%	7%	6%
	16157	9141	56.57%	53%	22%	11%	8%	6%

VII. CONCLUSIONS

In this document we explored recommender systems for visual programming environments. We looked at previous

work performed in the field of recommender systems both for text- and graph-based environments. We then proposed an implementation of a recommender system in the Blueprint environment found in Unreal Engine 4. Using an implementation of the proposed recommender system we then performed an evaluation in the form of user tests and an automated test. From the automated test we can see that with the current approach we are able to provide a correct suggestion in a 5-entry short-list in 73% of the situations. From timing measurements obtained from the user tests we can see that the current system does not help productivity and in our test case on average slightly slowed the users down (4%). This could be caused by the user test participants being unfamiliar with the system or the system being rendered useless by the specific workflow they had previously developed while working in Blueprint. In spite of the decreased productivity the perceived productivity increased when using the suggester system as indicated by the user survey.

VIII. Further Research

A. Wisdom of the Crowds

Currently all suggestions have to come from the local database of Blueprints found in the project of the user. Koop et al. suggested that the wisdom of the crowds could be leveraged to generate suggestions from the community [15]. A way of implementing this is building a centralized server that receives suggestions from all clients that use the plugin and in turn serves suggestion from all accumulated suggestions. An interesting field of research might be to explore the effect on the quality of suggestions by using such a system.

B. Localized cache

Another addition might be the addition of a localized cache. As Tu et al. discussed every programmer has their own programming style [11]. It might be worthwhile to explore whether or not this is also the case in a VPE. By creating two suggestion caches: a global cache built from all available graphs and a local cache from the current Blueprint accuracy might be increased. Suggestions including graph-local variables can also be stored in the local cache to avoid suggestion interference in other graphs.

C. User tests

Further user testing has to be performed to give more accurate results about the impact a suggester system has on workflow. Due to time constraints the user tests performed in this research was limited in scope. To fully investigate the impact of such a system a longer test would have to be performed where the participants have ample time to adjust and incorporate the suggestions in their workflow.

D. Pin differentiation

The current implementation creates the nodes based on datatypes only. An improvement to this might be to also take the pin name into consideration while generating and querying suggestions. This will allow the system to differentiate between pins of the same type on a node. This might lead to more accurate suggestions but could reduce the amount of suggestions that can be found for a particular situation. It might be interesting to see if this trade-off improves the quality of suggestions made by the suggester system.

References

[1] I. Epic Games, "Unreal engine technology," 2015. [Online]. Available: https://www.unrealengine.com/

[2] G. Crytek, "Cryengine," 2015. [Online]. Available: http://cryengine.com/

[3] A. Nguyen, T. Nguyen, H. Nguyen, A. Tamrawi, H. Nguyen, J. Al-Kofahi, and T. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," *Proceedings of the 34th International Conference on Software Engineering*, pp. 69–79, 2012. [Online]. Available: http://ieeexplore.ieee.org.cyber.usask.ca/xpl/articleDetails.jsp?tp=&arnumber=6227205&contentType=Conference+Publications&searchField=Search_All&queryText=Graph-based+pattern-oriented,+context-sensitive+source+code+completion

[4] M. Bruch, M. Monperrus, M. Mezini, and L. Briand, "Learning from examples to improve code completion systems," *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E*, p. 213, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1595696.1595728

[5] R. Robbes and M. Lanza, "How program history can improve code completion," *ASE 2008 - 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings*, pp. 317–326, 2008.

[6] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn, "CACHECA: A Cache Language Model Based Code Suggestion Tool," *ICSE Demonstration Track*, 2015. [Online]. Available: http://www.zptu.net/papers/icse2015_cacheca.pdf

[7] Microsoft, "Using intellisense," 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx

[8] B.-j. P. Hsu and G. Ottaviano, "Space-Efficient Data Structures for Top- k Completion," *Proceedings of the 22Nd International Conference on World Wide Web*, pp. 583–593, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2488388.2488440

[9] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. July 1928, pp. 379–423, 1948. [Online]. Available: http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

[10] K. O'hara, R. Harper, H. Mentis, A. Sellen, and A. Taylor, "On the Naturalness of Software," *ACM*

Transactions on Computer-Human Interaction, vol. 20, no. 1, pp. 1–25, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442106.2442111

[11] Z. Tu, Z. Su, and P. Devanbu, "On the Localness of Software," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 269–280, 2014.

[12] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool," *Proceedings - International Conference on Software Engineering*, pp. 1407–1410, 2012.

[13] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based Mining of Multiple Object Usage Patterns," *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp. 383–392, 2009. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595767

[14] F. T. D. Oliveira, L. Murta, C. Werner, and M. Mattoso, "Using Provenance to Improve Workflow Design," *Lecture Notes in Computer Science*, 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.6991

[15] D. Koop, C. E. Scheidegger, S. P. Callahan, J. Freire, and C. T. Silva, "VisComplete: Automating suggestions for visualization pipelines," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1691–1698, 2008.

[16] VisTrails, "Vistrails," 2015. [Online]. Available: http://www.vistrails.org/

[17] M. Bruch, T. Schäfer, and M. Mezini, "On evaluating recommender systems for API usages," *Proceedings of the 2008 international workshop on Recommendation systems for software engineering - RSSE '08*, p. 16, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1454247.1454254

[18] R. Kohavi, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," *International Joint Conference on Artificial Intelligence*, vol. 14, no. 12, pp. 1137–1143, 1995.