Dynamic User Interfaces via Incremental Knowledge Management

Michael Kohlhase[1], Richard Marcus[1], Navid Roux[1] and John Schihada[1]

[1]*Computer Science, FAU Erlangen-Nürnberg*

Abstract

Extending the UFrameIT Framework, we propose an upgrade to the User Interface features that is directly built on the knowledge-based part, MMT. In the game engine, it suffices to interpret the output and visualize it accordingly, resulting in features that work generically and out of the box for all games built with UFrameIT.

Initially, we were using fixed text descriptions and a static UI that created unnecessary mental load when players were trying to match between the abstract problem description and concrete game situations. To improve this, we introduce dynamic UI synchronization. We relate the progress of the player to the abstract formalized solution and give hints accordingly. These hints are generated via partial views; given a partial solution to the problem, MMT can check the dependencies and point towards the missing information. In the game, we can use this to highlight not yet assigned as well as even not yet existing facts in the game. This way, the player gets immediate feedback during the solution process.

Keywords

Serious Games, MMT, User Interfaces, User Experience, Knowledge Management, Development Tool, Unity, Game Engine

1. Introduction

The UFrameIT framework [CICM20] is a system and software library for supporting development and improving gameplay of knowledge-driven Serious Games. Serious Games range from educational games that teach about certain topics like math to process simulations like machine maintenance. However, for complex domains like STEM (science, technology, engineering, and mathematics), where practical application in a virtual world and gamification concepts promise engaging alternatives to traditional teaching methods, games are often equally difficult to develop.

The FrameIT Method We implement the FrameIT Method in the UFrameIT Framwork, which connects Unity [Uni] with the MMT system [Rab18], which takes care of managing

MathUI 2021

michael.kohlhase@fau.de (M. Kohlhase); richard.marcus@fau.de (R. Marcus); navid.roux@fau.de (N. Roux); john.schihada@fau.de (J. Schihada)
https://kwarc.info/kohlhase (M. Kohlhase); https://kwarc.info/people/rmarcus/ (R. Marcus); https://kwarc.info/people/nroux/ (N. Roux); https://kwarc.info/people/jschihada/ (J. Schihada)
0000-0002-9859-6337 (M. Kohlhase); 0000-0002-6601-6457 (R. Marcus); 0000-0002-8348-2441 (N. Roux); 0000-0002-8601-7449 (J. Schihada)

in-game acquired knowledge and makes use of the Mmt language to formalize this knowledge – helping game developers to focus on the actual game implementation.

We synchronize both sides via *facts*, pieces of information the player can collect by interacting with the game. To represent theorems, we use *scrolls*, which we also visualize parts of in the game so that players can interact with these (cf. Figure 2). In particular, players can access scrolls in the UI and enter facts into the scroll to form a solution to the problem in the game.

Players can access them in the UI and enter facts into the scroll to form a solution to the problem in the game.

Running Example The FrameIT Method is supposed to allow players to apply abstract knowledge in real-world problems that are presented in the form of Serious Games. As running example, we continue with the one from our original paper [CICM20], where we also describe the example in greater detail. The task is to solve a mathematical puzzle where the player needs to determine the height of a tree via trigonometry, e.g., using the tangent function (cf. Figure 1). In the game, the player marks the points at the base and the top of the tree, and an arbitrary third point on the ground. These points are added as *point facts* to the *fact store*. Then they measure two angles, the one enclosed at the root of the tree and the one on the ground, as well as the distance between the respective points. The resulting facts are then used to populate the scroll interface to produce the desired tree height as a fact.

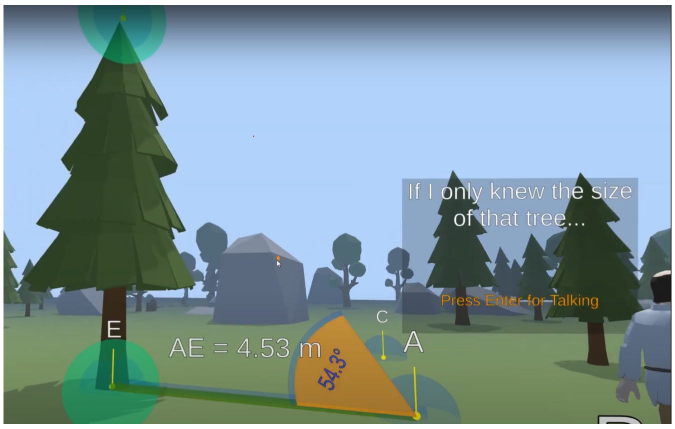
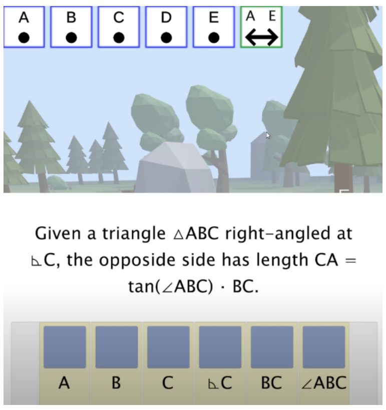


Figure 1: Running Example: Opposite Len: The player is supposed to compute the height of the tree by measuring facts and applying the tangent.

Problems of Unguided Solution Finding The big problem with our previous iteration was that players generally do not measure or compute the necessary facts in a fixed order. In contrast, textbook descriptions of applying the tangent function, for example, do use a certain set of variables to define the problem. Even with a good understanding of the topic in question that nudges players to measuring facts in the “best possible” order, assigning them correctly can cause problems. It may seem trivial to simply exchange these variables for applications to specific game situations, but players struggled to do this in practice because the mismatching variable names simply created additional mental load. Especially for novice players, it can then be difficult to discern if they are struggling with the content or just the user interface.

Let us look at at the basic situation depicted in Figure 2, where the player has measured a few facts (making up the triangle *DAE*) and is about to populate the so far empty scroll. If the player now assigned their facts labeled D, A, E to the facts A, B, C of the scroll, they would still see the static UI description of what the scroll is actually computing in terms of A, B, C. Importantly, the labels of the user-acquired points do not match the labels of the input slots required by the scroll (namely, A, B, C). Even more confusingly, the labels partially overlap since A appears in both sets of labels.

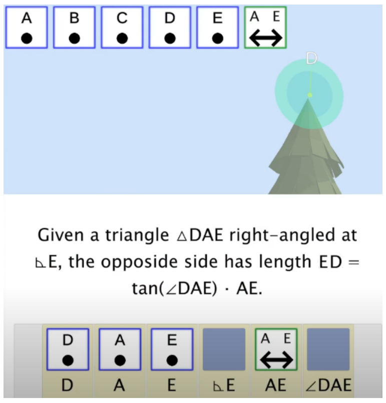


Given a triangle △ABC right-angled at ⊾C, the opposide side has length CA = tan(∠ABC) · BC.

Figure 2: Empty Scroll with Fact Collection at the top

An obvious remedy is to reflect the labels of the facts assigned so far in the labels appearing in the scroll description. This is one of the new features we will describe in this paper. However, this still can lead to ambiguous situations like the one depicted in Figure 3. The points and the distance are now assigned but which angles are required? The right angle can be identified directly but for the other angle *DAE*, it becomes important in which order the points of the triangle are to be interpreted. In three dimensions the player can even view the triangle from two different sides, invalidating traditional conventions. In this case, we can see that the point *D* is actually the point at the top of the tree (cf. Figure 3). Note that the difficulty here is then usually not the measuring of the correct facts in the first place but rather choosing the correct ones from the fact collection, in particular, when more facts than necessary have been collected.

Even ignoring the individual difficulties when trying to fill the scroll, the issue becomes apparent that the player only gets notified that the proposed solutions is incorrect after every fact has been assigned and the solution attempt has been tested by the system. Beside making sure that the type of fact is correct for each fact assignment, there has been no further user guidance feature so far.



Given a triangle △DAE right-angled at ⊾E, the opposide side has length ED = tan(∠DAE) · AE.

Figure 3: Partially Filled Scroll

While this can still seem rather manageable for the tangent example overall, moving to tasks that involve more variables, possibly with multiple substeps and scroll applications, further user assistance becomes vital, which we propose in the form of automatic knowledge-based hint generation.

Related Work We employ a separation of concerns of presenting (viewing, user-interfacing with) and processing (infering, deducing) knowledge. We outsource these concerns to UFrameIT as implemented in Unity and MMT, respectively. A similar separation is employed by many software products in the realm of graphical modeling. For example, tools like game engines (e.g., Unity), 3D modeling tools (e.g., Blender [Ble]), and graphical CAS-flavored systems (e.g., GeoGebra [GG]) all employ a similar separation. There, visualized objects (which we call facts) are often structured and accessible in a hierarchical *scene graph*.

In our approach and those of the mentioned tools, there is a bijective correspondence and synchronization between the visualized objects and the objects stored in the scene graph. This is an essential feature to allow users to perform operations on the scene graph that, e.g., the graphical frontend lacks an implementation for, and to perform vernier adjustments, e.g., of objects' coordinates.

In the UFrameIT framework, we go a step further and let the processing side also generate new objects/suggestions by means of inference and deduction. This is made possible by our usage of MMT as a generic knowledge management system. Moreover, via the API accessed by the presenting side, we communicate those objects to the player and let them play with as if they were objects the player had created themself.

Contribution Especially for serious games, it is important to give meaningful explanations to the player that adapt to the specific point of progress toward the solution. With MMT, the feature of showing hints emerges as an instantiation of our generic algorithms, i.e., it neither needs to be hardcoded for fixed in-game situations nor requires a custom crafted system monitoring the player's progress. In this sense, our contributions are the dynamic UI text synchronization, the automatic and generic generation of hints, and the necessary additions to MMT.

Note that in this paper, we want to present functionalities that can support professional developers in designing and implementing good UIs, and evaluating the quality of our current UIs would not be useful at this point. Our focus is on the pipeline within the UFrameIT framework that allows deriving UIs and related features directly from the knowledge representation.

Overview To lay the groundwork for the later description of the technical implementation, we will outline the MMT features that we make use of in the UFrameIT framework in Section 2. Section 3 describes the features from the player perspective so that we can show how they benefit during gameplay. We will continue in Section 4 by diving into the framework implementation and highlight the modular approach between Unity and MMT. Section 5 concludes the paper and gives an outlook on possible future framework extensions.

2. FrameIT Knowledge Management in MMT

Here we will summarise the basic MMT foundations and concepts that our base framework is built on.

MMT The MMT system offers a framework for flexiformally representing and reasoning with declarative languages such as logics, type theories, and set theories. This makes it also suitable to represent and reason with a wide range of mathematical knowledge, as desired for our purposes. Flexiformality allows formalizations in a way that is formal enough for applications (such as FrameIT) to depend upon, but still informal enough to obviate the need to derive everything from first principles (e.g., as is common in proof developments in Coq). The MMT API implements complex algorithms generically for any language in the framework. This is a crucial feature that makes the FrameIT approach scale to different use cases such as math-educational games (as in our running example) or physics-inspired simulations, e.g., to simulate work machines using cogwheels.

MMT Theories The primary way to organize knowledge in MMT is given by *theories*. These are effectively named lists of constants of the form $c\colon A[= t]$, where A is the *type* of c and t is an optionally given definition. Here, the terms A and t stem from some user-chosen dependent type theory. Since we skip over exact details of representation anyway and keep developments in this paper at an intuitive human-readable syntax, we omit details of the type theory and refer to [Rab17]. Theories allow for modularity by means of **include** declarations.

Consider the following two MMT theories that formalize (some of) the background knowledge behind our running example:

$$
\begin{array}{l}
\textbf{theory } \texttt{OppositeLen}^P = \{ \\
\quad \textbf{include } \texttt{EuclidGeo} \\
\quad \texttt{A}, \texttt{B}, \texttt{C} \quad\;\; : \texttt{point} \\
\quad \texttt{AxAngleC} : \measuredangle\texttt{BCA} \doteq 90^\circ \\
\quad \texttt{AngleB} \quad : \measuredangle\texttt{ABC} \\
\quad \texttt{DistBC} \quad : \overline{\texttt{BC}} \\
\}
\end{array}
\qquad
\begin{array}{l}
\textbf{theory } \texttt{OppositeLen}^S = \{ \\
\quad \textbf{include } \texttt{OppositeLen}^P \\
\quad \texttt{DistCA} : \overline{\texttt{CA}} \doteq \tan(\texttt{AngleB} \cdot \texttt{DistBC}) \\
\}
\end{array}
$$

The theory $\texttt{OppositeLen}^P$ first includes a development of Euclidean Geometry (omitted) and then goes on to declare a handful of undefined constants. In general, if a theory contains undefined constants, we can think of it as an abstract interface (as in software engineering) whose implementation details are still outstanding. Here, $\texttt{OppositeLen}^P$ captures the abstract interface of a triangle (given by $\texttt{A}, \texttt{B}, \texttt{C}$) with one known side length and an adjacent angle (given by `DistBC` and `AngleB`) and that is right-angled at `C` (demanded by `AxAngleC`). In such an abstract situation, mathematics tell us that the length of the side opposing `DistBC` and `AngleB` can be computed via $\tan(\texttt{AngleB} \cdot \texttt{DistBC})$. And this is precisely captured by the theory $\texttt{OppositeLen}^S$, which includes the abstract situation captured by $\texttt{OppositeLen}^P$ and adds one defined constant. This defined constant `DistCA` simply postulates that $\overline{\texttt{CA}}$ is given by this formula. Such a pair of $\texttt{OppositeLen}^P$ and $\texttt{OppositeLen}^S$ is exactly what we call scroll. The UFrameIT Framework organizes each fact – be it from the player marking an object in the virtual world or generated by MMT – as constants into MMT theories.

MMT Views Concrete instantiations of these abstract settings can be captured by *views*. Every view is a truth-preserving translation from some domain to some codomain theory by assigning for every undefined domain constant a codomain expression. Suppose `ConcrSituation` is a theory including `EuclidGeo` and containing three points $\texttt{P}, \texttt{Q}, \texttt{R}$. For example, this theory could have been built by the player marking those three points in the virtual world. We can now instantiate $\texttt{OppositeLen}^P$ in `ConcrSituation` by the view v from the former to the latter (given below). Importantly, this allows us to carry over $\texttt{OppositeLen}^S$ to the concrete situation, too, by means of a pushout in the category of theories and views. The resulting theory `NewSituation` is presented in Figure 4. Looking at this from the game perspective, such a pushout happens when players apply the scroll, while the view itself phrases a specific game situation abstractly.

$$
\begin{array}{l}
\textbf{view } v\colon \texttt{OppositeLen}^P \to \texttt{ConcrSituation} = \{ \\
\quad \begin{array}{ll}
\texttt{A} & = \texttt{P} \\
\texttt{B} & = \texttt{Q} \\
\texttt{C} & = \texttt{R} \\
\texttt{AxAngleC} & = \text{some proof of } \measuredangle\texttt{QRP} \doteq 90^\circ \\
\texttt{AngleB} & = \text{the value of } \measuredangle\texttt{PQR} \\
\texttt{DistBC} & = \text{the value of } \overline{\texttt{QR}}
\end{array} \\
\}
\end{array}
$$

$$\begin{tikzcd}
\texttt{OppositeLen}^P \arrow[r, rightsquigarrow] \arrow[d, hook] & \texttt{ConcrSituation} \arrow[d, hook] \\
\texttt{OppositeLen}^S \arrow[r, rightsquigarrow] & \texttt{NewSituation} \arrow[ul, phantom, "\ulcorner", very near start]
\end{tikzcd}$$

$$
\begin{array}{l}
\textbf{theory } \texttt{NewSituation} = \{ \\
\quad \textbf{include } \texttt{ConcrSituation} \\
\quad \texttt{DistRP}\colon \overline{\texttt{RP}} \doteq \tan(\text{the value of } \measuredangle\texttt{PQR} \cdot \text{the value of } \overline{\texttt{QR}}) \\
\}
\end{array}
$$

Figure 4: The `NewSituation` Theory as a result of a pushout

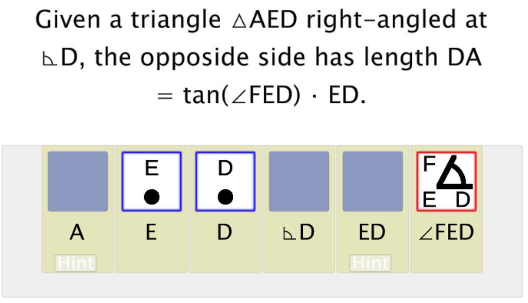
Applying Scrolls Therefore, with the application of scrolls and the respective pushout computation, new knowledge/facts can be gained from existing ones. Intuitively, the pushout `NewSituation` can be seen as the union of `ConcrSituation` and `OppositeLen`^{S}, so that they exactly share `OppositeLen`^{P} [Rab]. Such a pushout exists, if the provided view v is complete, so that each constant in `OppositeLen`^{P} is mapped onto a constant in `ConcrSituation` and the single types of the mapped components match. However, the basic view functionality is not sufficient for the dynamic UI features we need in UFrameIT games. For the following dynamic UI features, we need to check views more than once at the end and need further additions to the framework.

3. Player Guidance through Dynamic User Interfaces

To alleviate the problem of missing player guidance in UFrameIT games, we introduce *Dynamic Scroll Descriptions* and *Visual Hints*. Both of these together make up our new Dynamic User Interface, which allows players to get help from the system step-by-step.

With these, we can continuously match the problem description with the solution proposed by the player (cf. Figure 5). This feature is very straight forward. The player can insert facts that were collected through interaction with the game world into the slots within the scroll UI that are required to solve the game problem. Since the problem description in the UI refers to exactly these required facts, the system just needs to update the variables in the description text.

Dynamic Scroll Descriptions This way, the player always knows which place assigned facts have in respect to the solution, e.g. the measured



Given a triangle △AED right-angled at ⊾D, the opposide side has length DA = tan(∠FED) · ED.

Figure 5: Scrollview Assistance

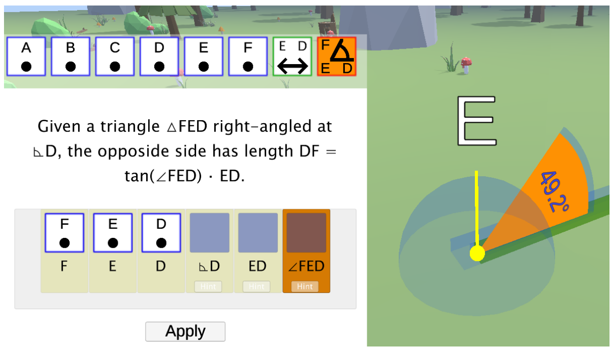
point *D* is the point of the triangle where a right angle needs to exist.

On the MMT side, we have implemented this in the form of a templating engine, which allows to interpret parts of the scroll description as links to the facts that are used in the scroll formalization.

Visual Hints Our system can automatically generate hints for next steps during the solution process (cf. Figure 6) With this feature, players can ask the system to derive the next step based on incomplete solutions. A simple example might be that the player starts with assigning the measured distance between a point on the ground *E* and the root of the tree *D*. For this to be a possible solution to the problem, the second and third point in the scroll need to be exactly these points. Consequently, requesting a hint in this state highlights the slots in the scroll, the suggested two points in the fact collection list and their corresponding fact representation in the game. We call this *backward view completion*. Vice versa, assigning the three points that are supposed to make up the triangle also define the angles and the distance that are necessary to apply the tangent. The system can then show the player where the respective facts are by using an animation, equivalent to the one used for backward view completions. In contrary to those, we can now have the special case, that the respective fact isn't already existent. Nevertheless the game-side temporarily creates and animates a new fact with the same properties. We call this *forward view completion*.

To compute these in MMT, we use partial view applications: we now allow implicit assignments in views, which allow us to compute missing dependencies as we now longer have to input all defined facts of a scroll for the view.

Especially the visual hints via view completions can directly lead the player to the solution, lowering the difficulty of the game and possibly preventing players to come up with the solution themselves. In this sense, didactic experts or developers are supposed to define the circumstances when the hints should appear. One way to do this might be to create separate tutorial and application modes but more fine-grain variants are also possible.



Given a triangle △FED right-angled at ⊾D, the opposide side has length DF = tan(∠FED) · ED.

Figure 6: Hint Animations

Note that in a software like this, which tries to support the develop process, possible benefits of using it can be viewed from two angles:

- The end-user perspective: how do they benefit from the supported features?
- The developer perspective: how much work can be saved?

The important point for our framework is that developers essentially get these features for free and only have to configure them to their preferences. We will describe the implementation that allows this in the next section in detail.

4. Enabling Dynamic User Interfaces

To enable the described UI features in a way that can be applied to future, possibly very different game problems or domains, we need a generic integration. This is exactly where we exploit the fact that we use a knowledge based backend in the form of MMT. In the following, we will describe the MMT implementation and the synchronization with the game engine.

Synchronizing Managed Knowledge with the User Interface In UFrameIT, synchronization is a bidirectional process where the current state of knowledge about the virtual world is exchanged between the game-side and MMT. Until now this synchronization took place in the following situations: [CICM20]

1. List available Scrolls: The game-side queries, which operationalized mathematical theorems (scrolls) MMT can offer
2. Add facts: The game-side tells MMT, which facts the player has acquired via gadgets
3. Request Pushout computations: The game-side transmits a complete scroll mapping, MMT tries to derive new knowledge and correspondingly responds

In order to establish dynamic game user interfaces, another synchronization step after each scroll mapping had to be integrated, where MMT already tries to derive new knowledge from partial scroll mappings. Consequently, the derived knowledge can be used as player guidance or assistance.

```
{
    "scroll": "MMT URI",
    "assignments": [
    {
        "fact": {"uri": "MMT URI"},
        "assignment": /* SOMDoc */
    },
    ...
    ]
}
```

Listing 1: Scroll Application Format

The new server endpoint which serves that purpose receives json-data in the same format as in synchronization step in 3. In both cases, a scroll reference and its corresponding scroll mappings are needed, see Listing 1. Both scrolls and facts can be referenced by means of MMT uri's. Furthermore the assignment list consists of elements that represent the mapping between the abstract fact-element of the scroll and a concrete fact acquired inside the game. The 'SOMDoc'-Format used for the 'assignment'-property in Listing 1 is a result of the complete rework of the overall json format structure employed in the frameit-server. This format is used in most of the server endpoint data and allows us to express almost any MMT symbol declaration, facts included. 'SOMDoc' is a JSON representation of an OMDoc subset [Koh05] and implements several parts of the OpenMath-JSON standard [Wie].

```
{ "kind": "OMI", "decimal": 0 }
{ "kind": "OMF", "float": 0.0 }
{ "kind": "OMS", "uri": "MMT URI" }
{ "kind": "OMSTR", "string": "string" }
{
    "kind": "RAW",
    "xml": "OMDoc XML string"
}
{
    "kind": "OMA",
    "applicant": /* SOMDoc */,
    "arguments": [/* SOMDoc */]
}
```

Listing 2: SOMDoc Format

Listing 2 shows a list of all possible 'SOMDoc' json terms, where the 'RAW' term is a way to encode unrepresented terms, which are excluded from the represented OMDoc subset. After MMT has processed the request payload, the FrameIT-Server is able to fill the dynamic scroll info data (Listing 3) and send it back to the game-side.

Figure 7 shows an example, where the player selected three points P,Q and R and measured the angle $\measuredangle PQR$ between them. Afterwards he mapped the angle $\measuredangle PQR$ onto the scroll's abstract angle $\measuredangle ABC$. As both $\measuredangle PQR$ and $\measuredangle ABC$ are Angle-Facts, which internally reference three Point-Facts P,Q,R and A,B,C respectively, the server is able to infer that these points should be mapped onto each other. Therefore these types of insights get wrapped into the 'completions'-field (Listing 3) in the same format as the assignments from Listing 1.

The rendered-field (Listing 1) is another way to guide the player for finding the correct scroll mappings, especially when scrolls become more complex and the amount of components increases. Each scroll (Listing 4) has a description and a list of required facts that are necessary to compute the acquired facts. Each time the player changes the scroll mapping, the returned rendered scroll will change and affect the scroll's text labels that are visible in the game UI. Not only the required fact's label will adopt the one from the assigned fact, also the description internally consists of sub-labels that get correspondingly adjusted (see Section 3).

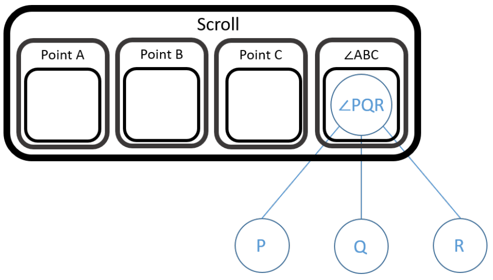


Figure 7: Backward-Completions

Furthermore the required facts represent the inference result in forward direction. Figure 8 depicts the situation where a player measured the same facts as in Figure 7 but instead of mapping angle $\measuredangle PQR$ onto angle $\measuredangle ABC$, he only mapped its components P,Q,R onto A,B,C. As a consequence the required fact $\measuredangle ABC$ of the rendered scroll internally references the already assigned facts P,Q,R. The player can therefore be advised to additionally map an AngleFact that represents an angle between the components P,Q,R onto angle $\measuredangle ABC$. In order to provide the game-side with a complete overview of the current state of knowledge, the two remaining fields 'valid' and 'errors' indicate if the sent scroll mappings are complete, correct and sufficient for potentially computing pushouts. Even though they are currently not processed from the game-side, they can be of special interest for upcoming features.

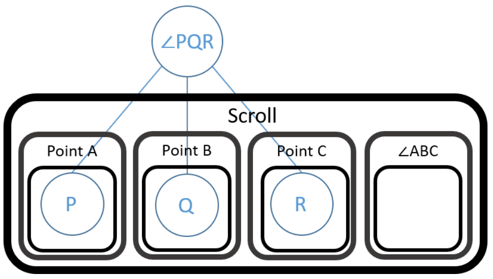


Figure 8: Forward-Completions

```
{
    "original": /* Scroll */,
    "rendered": /* Scroll */,
    "completions": /* Assignments */,
    "valid": true|false,
    "errors": /* Errors */
}
```

Listing 3: Dynamic Scroll Info Format

```
{
    "ref": "MMT URI",
    "label": "label",
    "description": "description",
    "requiredFacts": [/* Fact */],
    "acquiredFacts": [/* Fact */]
}
```

Listing 4: Scroll Format

Integrating Managed Knowledge into the User Interface In order to integrate the managed knowledge and therefore obtaining a dynamic user interface (see Figure 6), the game-side has to process the dynamic scroll info data (Listing 3) returned from the endpoint. The Dynamic Scroll Descriptions (see Figure 5) are simply generated from the labels of the rendered scroll and the corresponding required facts (see Listing 3,4). As both scrolls and facts do have a reference, the counterpart in the UI can easily be looked up and adjusted. However, the establishment of visual hints in the UI is split up into two parts:

1. Enabling hint buttons for specific scroll parameters
2. Performing visual hints when the corresponding buttons are pressed

In part 1, we want to find out for which scroll parameters hints are available while already preparing data for part 2. Therefore, both the forward- as well as the backward-completions, reflected by the completions- and the rendered-field (see Listing 3), have to be examined. For the completions-field the search process is straight forward: Each completions-assignment references a scroll parameter and an associated fact, that has already been determined by the player. Thus, the hint-button for the corresponding scroll parameter is enabled and the assignment is stored for potential, subsequent hint invocations. For the rendered-field, we're only searching for complex, parsable required facts that haven't already been assigned. In this case, each existing fact type that internally references other facts is meant to be complex and every complex fact where these referenced facts already exist is parsable. If we found some of these we can once more enable the corresponding scroll parameter and save the parsed fact. As the data for part 2 is already prepared, the remaining steps are very simple:

1. Find an entry in the prepared data, that references the requested scroll fact
2. Find the fact in the global fact-list, that matches the prepared data entry
3. If no matching fact was found in step 2: Create a temporary fact with the same properties, that destructs itself after the hint animation
4. Animate the requested scroll parameter
5. Animate the representation of the fact, found in step 2/3, in the game
6. Animate the fact icon in the fact collection list, that belongs to the fact, found in step 2/3

5. Conclusion

We have presented two additions to the UFrameIT framework that make user interactions in UFrameIT games much more fine-grained and intuitive. *Dynamic Scrolls Description* dynamically updates the variables names in scroll formulae to the assignments already fixed by the user. *Automatic Hint Generation* allows to visualize scroll components in the game world by highlighting them selectively.

With these new features players are now guided through the game with the help of a dynamic UI that can deliver hints to the player and synchronize variable names. We have implemented generic algorithms in the MMT system that the UFrameIT framework instantiates to get those features. Moreover, we have augmented the UFrameIT API as little as possible such that the game engine only interprets the information sent to it by MMT and thus is generic in improvements on the MMT side.

The UFrameIT framework is under continuous development, and we are aiming for a knowledge-based serious game development toolkit. Next steps will include functionality for crafting in-game problems like the running example in the game itself and assembling such problems into dynamic game levels. For both we also want to make use of the knowledge-management capabilities of MMT: for problems, we can either search for 3D configurations that fit a given scroll, or find matching scrolls given a problem. For level aggregation, we can use the MMT (background) knowledge graph to select "next problems" where the user has already mastered all but one prerequisite in earlier parts of the level/game.

The work presented here constitutes the base for a game developers' API that allows fine-grained customization of the set of available features (e.g., hints, scrolls, in-game problems), thus allows developers to flexibly craft their own games while obviating much of the overhead our framework takes care of.

References

[Ble] Blender. *Blender: the free and open source 3D creation suite.* https://www.blender.org/. Version 2.93. URL: https://www.blender.org/ (visited on 07/14/2021).

[CICM20] Michael Kohlhase et al. "FrameIT: Detangling Knowledge Management from Game Design in Serious Games". In: *Intelligent Computer Mathematics (CICM) 2020.* Ed. by Christoph Benzmüller and Bruce Miller. Vol. 12236. LNAI. Springer, 2020, pp. 173–189. ISBN: 978-3-030-53518-6. URL: https://kwarc.info/kohlhase/papers/cicm20-frameit.pdf.

[GG] International GeoGebra Institute. *Graphing Calculator – GeoGebra.* URL: https://www.geogebra.org (visited on 05/27/2020).

[Koh05] Michael Kohlhase. "OMDoc: Open Mathematical Documents". In: *Open Source for Education in Europe: Research and Practise.* Ed. by Fred de Vries et al. Proceedings at http://hdl.handle.net/1820/483. Open Universiteit Nederland. Heerlen, The Netherlands: Open Universiteit Nederland, Nov. 2005, pp. 137–143. URL: http://hdl.handle.net/1820/483.

[Rab] Florian Rabe. “Theory Expressions (a Survey)”.

[Rab17] Florian Rabe. “How to Identify, Translate, and Combine Logics?” In: *Journal of Logic and Computation* 27.6 (2017), pp. 1753–1798.

[Rab18] Florian Rabe. “MMT: A Foundation-Independent Logical Framework”. Online Documentation. 2018. URL: https://kwarc.info/people/frabe/Research/rabe_mmtsys_18.pdf.

[Uni] Unity Technologies. *Unity Realtime Development Platform.* https://unity.com/. Version 2019.3.6. URL: https://unity.com/ (visited on 03/19/2020).

[Wie] Tom Wiesing. *OpenMath-JSON.* URL: https://omjson.kwarc.info/ (visited on 10/13/2018).