Exploring the Immediate Mode GUI Concept for Graphical User Interfaces in Mixed Reality Applications

Felix Brendel, Sven Liedtke

Technical University of Munich

Abstract: The state of the art GUI is based on widget lifetimes that are bound to objects or handles. This common approach which is called retained mode GUI has a few drawbacks. The application data has to be kept in sync with the UI manually. An alternative approach is the immediate mode GUI which differs from retained mode in that no UI widget has a lifetime and does not require updates. In every frame, the application instructs the immediate mode system what to display on the screen. Immediate mode GUI systems are already widely used in games and game tooling. In this work, the possibilities of an immediate mode system in the context of mixed reality are explored.

Keywords: Mixed Reality, Graphical User Interfaces, Immediate Mode, Retained Mode

1 Introduction

Two famous examples of mixed reality (*MR*) are augmented reality (*AR*) and virtual reality (*VR*). MR user interfaces (*UIs*) are no longer 2D; they can be positioned in 3D space or even be attached to the users' body parts, such as a "hand menu" in AR. However, these input devices are not well suited for all kinds of MR applications. With VR, the input devices are invisible to the user as they only perceive the virtual scene.

Since the UI can be part of the virtual scene in MR, it is desirable that, when multiple users take part in the same virtual environment, the state of shared user interfaces be kept in sync between all the users. Especially in AR the UI might undergo many transitions. Maybe some interaction possibilities should only become visible when the user gets close to them to avoid their view being cluttered. Additionally, UI elements might have different ways of displaying themselves. A clock widget might be shown on the user's wrist if it is visible or otherwise, it might be shown as a screen-mounted head-up display (*HUD*).

Imagine for instance a mixed reality interior planner. It would allow users to visualize what a room would look like with different kinds of furniture. They could use this application to plan the furniture layout of a room in their home. With augmented reality, they could stand in their room and see the virtual furniture around them. Furniture stores could offer this application in virtual reality using the size measurements of the room for which customers are looking for furniture. In this case, the UI would consist of textual descriptions of the shown items as well as object-mounted interfaces to modify the furniture. This could

be used to change the type of wood it is made of, or the shape of door handles. Additionally, the users have access to a UI that is created closely around them to create new furniture and place it in the room. Some information should always be visible and not be part of the 3D scene, like the total price or the number of all the placed items in the room. It should be shown as a screen-mounted HUD more towards the side of the peripheral vision to not obstruct the vision straight ahead.

The approach to bind widgets to objects or handles as done by the state of the art graphical user interface (*GUI*) libraries such as QT[1], wxWidgets[2] or tkinter[3] is called retained mode GUI (*RMGUI*). A contrasting approach is given by immediate mode GUIs (*IMGUIs*). They enjoy wide use cases in research publications and range from UIs monitoring metrics of real-time distributed computations for CERN LHC [Fus21] over special purpose visualization UIs for code architecture [OEW19] to GUI environments to edit and analyze complex boolean functions [PŠ20]. Outside of research, information about implementation details of commercial software is usually not available with the exception of open source projects.

Until today, one of the only publications further dealing with the IMGUI concept itself is the work of van Bernem. They show the viability of IMGUIs for traditional desktop applications and implement "NBUI", a prototypical single header IMGUI library [vB19]. Performance wise it is on par with existing RMGUI implementations while offering flexibility regarding widgets' behaviors and hot reloadable layout files that are used to design the UI in a declarative way. Due to the lack of further literature and the fact that the authors of IMGUI systems tend to work in the industry rather than academia, the only way to get more information about their systems is through studying their source code or more informal media like conversations or recorded demonstrations.

Motivated by the absence of a formal analysis of the IMGUI approach this work aims to examine existing IMGUI implementations. The components of an IMGUI system implementation are explained and variations of them are discussed in terms of their implications and trade-offs. Each component is also brought into the domain of mixed reality. This work will put focus on the needed changes to the core concepts of IMGUI to be usable as an MR IMGUI.

The first appearance of IMGUI was informally published by Casey Muratori in 2005 in form of a YouTube video [Mur05]. All widgets that should be visible are communicated to the GUI system every time the GUI is drawn. Sean Barret wrote an article about IMGUI in the Game Developers Magazine as one of the first formal resources on IMGUI in 2005. He describes RMGUI as a "premature optimization" of a more simple IMGUI system because the additional mental overhead of widget lifetime management is introduced [Bar05]. In what is considered to be one of the first GUI implementations for the Xerox Alto, a retained mode GUI was implemented. The author Alan Kay acknowledged that directly rendering the data every frame is simple and powerful, but he was unable to do so with the

[1] `https://www.qt.io/`

[2] `https://www.wxwidgets.org/`

[3] `https://docs.python.org/3/library/tkinter.html`

available computational power at that time [Kay96]. Although Casey Muratori coined the term IMGUI, he was not the first one to use them. The 3D creation suite Blender[4] uses an IMGUI system for its GUI. The documentation for Blender's GUI system dating back to 2003 explains that each time the window is redrawn, all UI elements have to be created again. No UI widgets persist between redraws [Roo03].

2 Implementation of an IMGUI System for Mixed Reality

In particular, an IMGUI library should place the UI widgets in the scene and react to input. At the same time, the application programmer should not need to be aware of the state of the GUI as it is managed only internally in the IMGUI library. This sets it apart from RMGUI systems. The widely used general purpose IMGUI libraries like Dear ImGui[5], Nuklear[6], egui[7] and microui[8] show similarities in their architecture, which can be adapted to MR.

IMGUI applications, by their nature of querying input and communicating the whole UI state each update cycle to the GUI system, are frame based, similar to mixed reality applications. In the interior furniture planner example, the application would create all UI elements each frame as they should be shown. This allows for flexible behavior. If the furniture-mounted menu should only be shown to users in proximity to the furniture piece, the application simply calculates the users' distances each frame to determine if the menu should be shown. Only a small indication will be shown if the users are far away. This stands in contrast to an RMGUI library where for each frame it has to be decided how new widgets have to be created or the existing widgets have to be updated or deleted, based on the widget handles that the programmer has to manage.

The user's input has to be processed to be able to respond to events and the currently visible widgets have to be communicated to the rendering backend. This communication is based on geometric primitives that allow the rendering of text images, which make up all of the UI, taking into consideration additional styling information for the widgets. Each UI widget is internally identified with a unique ID, which the UI system can use to store additional state for widgets. The position and size of the widgets are determined by the layouts used. Simple layouts only require a single processing pass over the widgets, while others require multiple passes.

2.1 Input Handling

The way the user input is exposed to the application differs between platforms. The aim of the input handling system is to offer the rest of the library a common interface to query the users' input. While for non-binary inputs like mouse positions in traditional UIs or fingertip

[4] https://www.blender.org/
[5] https://github.com/ocornut/imgui
[6] https://github.com/Immediate-Mode-UI/Nuklear
[7] https://github.com/emilk/egui
[8] https://github.com/rxi/microui

positions in AR UIs, it is sufficient to supply the library with the current position each frame. For digital inputs like keys or buttons which are often used on VR controllers, relevant info is concerned with the change of button states between frames.

Only reporting the interactions based on the last frame and the current frame can lead to unregistered input interactions. If a key was pressed this frame and the user releases and presses it again before the frame ends, a simple input system would not register this interaction. This is because the button was pressed both times when the input was queried. With frame times as small as $\frac{1\text{ sec}}{60\text{ frames}} = 16.\overline{6}\frac{\text{ms}}{\text{frame}}$ for 60 FPS or $\frac{1\text{ sec}}{90\text{ frames}} = 11.\overline{1}\frac{\text{ms}}{\text{frame}}$ for 90 FPS, which is a common goal for VR applications [Hec16] this is less of a concern.

For a correct input handling system, the input system stores the current state of all keys or buttons – pressed or not pressed – together with the number of how many state transitions this key or button underwent in this frame. This can be done by either receiving input messages from a message queue as it is done in Windows [Mic21a] or by registering callbacks for the button events that modify the table. Like this, multiple input events concerned with one key or button during one frame can be handled. The input system would internally manage a table that maps each button and key to its current state and the number of state transitions that occurred during this frame. This table is then used by the rest of the IMGUI library, which depends on the updates.

2.2 Rendering

Since the common IMGUI libraries aim to be renderer independent, so that they can be integrated into any existing rendering system, they do not directly interact with the graphics hardware. Instead, for each widget that the application requests the IMGUI to draw, it will record the necessary drawing actions in platform independent *draw lists*. At the end of each frame, the application requests a summary of what to draw on the screen from the GUI library consisting of these draw lists. The contents of these draw lists then have to be translated to actual draw commands for the targeted platform.

Libraries vary in the level of abstraction in their draw lists. While Dear ImGui uses draw lists consisting of vertices and indices together with texture IDs, Nuklear chooses higher level primitives such as lines, curves and triangles. If the application already has the means to render geometric primitives like the ones emitted by Nuklear, it can directly display them. These high-level draw commands can optionally also be converted to a low-level draw list consisting of a vertex buffer and draw commands indexing into the vertex buffer with an attached texture ID [Met15]. This makes it suitable for efficient communication with GPUs.

Even in MR, where the user interfaces are not limited to the 2 dimensional screen space, the concept of the draw list fits the task well. Its spatial contents will then be 3D, while HUD contents could be stored similarly to the 2D IMGUIs. The buttons for the furniture planner could then be 3D meshes like the state of the art MR buttons presented in the mixed reality toolkit, which can be shown as cuboids [mrt22b]. Low-level draw lists will still store vertices and indices, while higher-level draw lists would be 3 dimensional drawing primitives.

2.3 Images and Text

Due to the concept and structure of the draw lists discussed in subsection 2.2, images are rendered as rectangles. They are associated with texture coordinates and the texture ID of the image to be rendered. The job of the IMGUI system is solely to perform the *layouting* (assigning space to all the widgets according to some rules) and creating the relevant entries in the draw lists.

To render text, a *font atlas* is used. It is an image texture containing all the glyphs in a given font that the application expects to be using. To help create a font atlas, the single header stb libraries[9] contain `stb_truetype.h`, an open source implementation for rendering font glyphs to a texture. Determining the extent of the layouted text is not a cheap computation as the dimensions of each displayed character have to be fetched from the font atlas, resulting in $\mathcal{O}(n)$ lookups for a string of length n.

Text rendering in MR benefits from having font atlases for at least two font sizes. To avoid blurry text, one font atlas will be optimized for the font size used in the 2D HUD, while the other one will be used for text in the 3D user interfaces. Except for having multiple atlases for different kinds of text, the approach is similar to traditional 2D UI. With this, the furniture planner application will be able to render text on the buttons and display the total price and number of furniture pieces in the HUD. For that, it will create little meshes that when rendered show the glyphs from the font atlas texture that represent the text to show.

2.4 Widgets

In an immediate mode, GUI widgets are created and recreated every frame. No lifetimes persist between frames. The simplest way to achieve this is by creating the widgets through function calls. Each widget has its corresponding function that, when called, will result in the widget appearing in the next frame. This is done akin to the immediate mode rendering of 3D objects in the scene. These functions serve two purposes. For one, they register and layout the widget in the GUI system for the upcoming frame. They also check how the user interacted with the widget in this frame and communicate the interactions that occurred via their return value. The interactions are determined based on the layouted button position and dimensions and the input that occurred this frame.

Muratori introduces the concept of `hot` and `active` widgets. A widget is considered `hot` if the user is about to interact with it [Mur05]. For widgets like buttons or sliders, this is the case if the mouse cursor or pointer in VR is hovering over the widget. A widget will stay `hot` as long as its hot-condition is met. Additionally, as soon as the user starts interacting with a widget (for example by pressing the left mouse button or starting an interaction on the VR controller) it will also be considered to be the `active` widget. The widget will stay `active` as long as the user is holding the button down. In terms of `hot` and `active`, a button is

[9] `https://github.com/nothings/stb/`

considered clicked if the mouse button was released while it is both `hot` and `active`.

To be able to internally manage the `hot` and `active` elements, each widget is associated with a unique widget ID. This ID is expected to remain constant between frames. A possible implementation would require the application programmer to uniquely name all of the used widgets. This can be cumbersome and error prone. Instead, the common IMGUI libraries create an ID based on the provided attributes of the widgets. The ID generation is usually based on a hash function that hashes the properties of the UI element. An extension to that API could allow the user to optionally provide any additional data, numbers or strings, which are then taken into account when the widget ID gets computed to disambiguate widgets with similar attributes.

An IMGUI library for MR cannot only store one global `hot` and `active` item as is sufficient for a 2D UI, because the user can be interacting with multiple elements simultaneously. For each input entity that can cause an interaction with the user interface, a separate `hot` and `active` has to be stored. In the case of AR, users might interact with the UI with all ten of their fingers. To find the UI element that input devices like VR controllers point to, a ray or parabola can be cast from the input device into the scene [mrt22a].

For AR, there is no discrete "hovering" state of the fingertip over a UI element. So a binary concept of `hot` might not be practical. The mixed reality toolkit provides different ways of showing how close the user is to interact with a widget. One way it does this is by using a proximity light. The user's fingertip emits light to highlight the `hot` widgets [mrt22b]. In the case where a widget is the `active` item of an input entity, it should not become `hot` for other entities as it is already being interacted with.

2.5 Internal State Management

Even though in IMGUIs no GUI state has to be kept in sync by the application programmer, IMGUI systems manage internal states of widgets. If a window is too small to show all its contents, it is common to display a scroll bar on the right side of the window. The position of this scroll bar is typically not communicated through the IMGUI API. The IMGUI API interaction would only consist of the creation of a fixed sized window and the creation of all its contents. The state of the scroll bar between frames is stored solely in the IMGUI library. The linkage between the widget and its internal state is based on its ID. If the user scrolls in the window with the name `configuration` then the scroll state is stored for its ID. When the GUI should be drawn next frame the scroll state for the `configuration` window can be retrieved. This is based on the idea that widgets have a fixed ID once created. Using the same technique, menus can be implemented where submenus can be expanded or collapsed. Each menu internally stores if it is collapsed or not.

An interaction with an IMGUI library when creating a menu might look like the following: The application creates a menu, supplying parameters for the library to be able to draw it. It might also supply a boolean value if the menu is initially expanded or collapsed. In the first frame, the GUI library encounters this menu widget (the ID was not in use yet), it will

create an internal state for it. It will then transfer the initial configuration of the widget (expanded or collapsed) to the internal state. For the upcoming frames, only the internal state is considered and is updated with the user's interactions. Once the widget is not drawn anymore, its internal state can be deleted. A simple way of garbage-collecting the unused internal states is to mark every internal state with a flag if it was accessed this frame.

2.6 Handling Layouts

Typically, the layout of the widgets is handled by the IMGUI system. While an API is possible where the application programmer is responsible for setting the widgets' absolute position and dimensions manually, the common IMGUI implementations layout the widgets automatically based on pre-defined layout styles inside panes or subwindows. These layouts are relevant for MR UIs as well as for traditional UIs. Most existing IMGUI implementations follow a simple approach, but more advanced approaches are explored in this section.

As simple layouts, the horizontal and vertical layout are introduced. In a horizontal layout, widgets are laid out from left to right while in a vertical layout widgets are laid out from top to bottom. These layouts can also be combined to create grid layouts. Inside a vertical layout, a horizontal layout can be created to be able to place widgets from left to right inside a row of the vertical layout. These layouts again focus on laying out widgets in pre-defined containers like panes or subwindows which in MR depend on the viewing direction and possible occlusions by other objects of the scene. Alternate approaches focus on layouting individual widgets in the scene for tasks like label placement [BFH01].

Non-adaptive layouts Virtually all existing IMGUI library implementations return the users' interactions to the application via the widget function return value. Each widget is processed only once per frame. These IMGUI systems can be classified as *one-pass IMGUI*. They have to layout the widgets on the fly in one pass where it is unknown how many further widgets will follow in the same layout.

Figure 1 depicts the structure of calls happening in a one-pass IMGUI in the form of a flame graph. The flame graphs in this section are only meant to visualize the structure of the API calls and are not meant to visualize their execution time with their horizontal extent. As a convention in this section, the functions written by the application programmer are shown with a light background, while the functions provided by the library are shown with a dark background.

To be able to return the users' interactions in a one-pass IMGUI, the implementation requires the layout of each widget not to be dependent on the widgets being created subsequently in the same frame. Conversely, this makes some layouts that depend on the number of widgets inside them impossible. For example, a seemingly trivial horizontal layout where widgets should be sized evenly fulfills this requirement and cannot be implemented trivially. There are still different possibilities to implement horizontal and vertical layouts in a one-pass IMGUI, albeit not adaptive ones. If widgets have a known size at creation time, they

can be laid out correctly at creation time and their position or dimensions do not depend on other widgets. In that case, the size of buttons would only depend on the size of the text label in them. For other widgets which do not inherently have a size, for example, by not containing text like a slider, the size of the widget would be explicitly provided. This approach is taken by the egui IMGUI library as well as Dear ImGui, where additionally the automatic size calculations can be overwritten with the desired widget sizes. Alternatively, if the widgets in a row or column should be evenly sized, the number of widgets per row or column must be known. With this information, it is possible to lay out widgets on the fly. For a horizontal layout, this means that for known n items per row and for each item $\frac{\text{container width}}{n}$ space would be allocated (in a simple case with no margin between the widgets). This is the approach that Nuklear and miroui chose for their API.

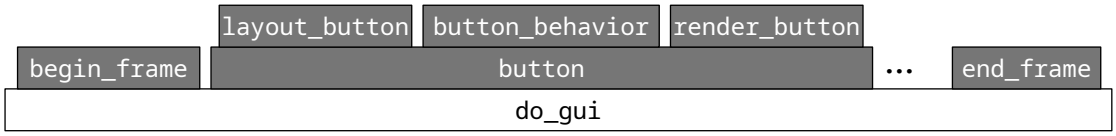

Figure 1: In a basic *one-pass IMGUI*, each widget already computes its final layout during its creation. It can therefore directly also respond to incoming input events (`button_behavior`) and add the relevant draw commands to the draw list.

Adaptive layouts As discussed above, one-pass IMGUIs cannot implement adaptive layouts where the position and size depend on the other widgets in the layout, which might not have been created yet. Without adaptive layouts, the application programmers would be limited in flexibility. To have equally sized buttons, the number of buttons must be known beforehand or a fixed size has to be given to all the buttons. This is the same for radial menus ("pie menus") where each button has to be explicitly supplied with a radial extent. In the furniture planner, not all furniture pieces would offer the same number of menu points. A menu point for changing door handles would only be visible for furniture with doors. With non-adaptive layouts, these factors have to be considered at the creation time of the GUI, putting more load on the application programmers.

The solution to adaptive layouts is to run multiple passes over the UI to first layout the whole UI before responding to the input events. This allows the application programmers to only focus on the content they want to show to the users and leave the layouting to the IMGUI library. There are different implementations: The *leapfrog passes* approach makes use of the completed layout of the previous frame to respond to input events for the current frame. Its schematic process would look similar to the one-pass IMGUI depicted in Figure 1, only with the position of `layout_button` swapped with `render_button`. In this approach, `layout_button` would prepare the layout for the next frame. This way, the IMGUI API can remain as simple as the one-pass IMGUI implementations with the added caveat, that input events are handled with the layout of the last frame which might have changed during this frame. Alternative approaches perform multiple passes during a single frame. One way to

achieve this is by running the GUI code multiple times per frame. In the *multiple explicit passes* approach, in the first pass, only the layout will be created and no input events will be handled. Then, in the second GUI pass, the input is considered with the finished layout and the GUI widgets report their interactions. A sketch of this process can be seen in Figure 2.

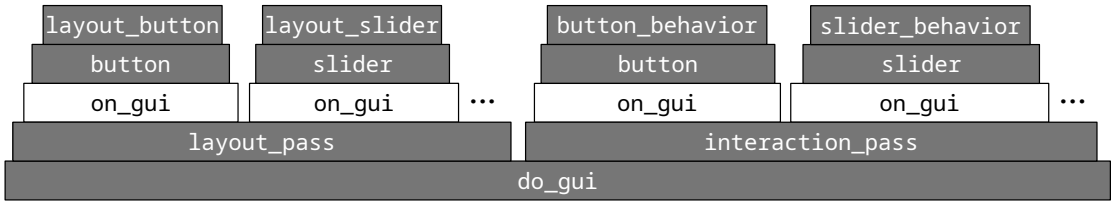

Figure 2: The *multiple explicit passes* approach. It is suited for scenarios where the GUI code is called by an external framework like a game engine. The `on_gui` functions depicted here can be seen as part of an entity script in an entity component system. In that case, multiple passes can be performed by letting the framework call the GUI code multiple times.

The technique is called *explicit* since the application programmer has to be aware that the GUI code will be called multiple times per frame. They have to be careful not to introduce side effects (that are not bound to UI events) in the GUI code, as they would be executed multiple times per frame. This is the approach taken by Unity's IMGUI system [Uni22]. To use the IMGUI system, users override the `OnGUI` method of the `MonoBehaviour` class. The documentation also mentions that the `OnGui` method may be called multiple times per frame. Using this method, Unity is able to implement adaptive layouts. If running the GUI code multiple times is not desirable, the *multiple hidden passes* approach can be used. Each GUI call is only recorded without doing any layouting or event processing yet. Only when the application tells the GUI library that the frame is finished the GUI calls are processed. At this point, the GUI library can perform multiple passes over the requested UI calls and can implement adaptive layouts. A schematic view of these actions can be seen in Figure 3.

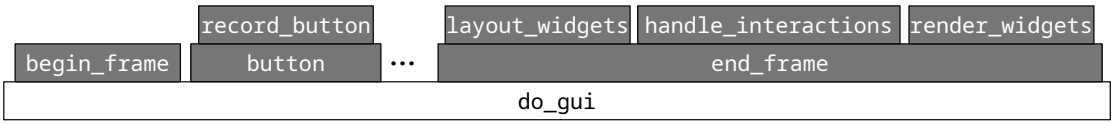

Figure 3: With the *multiple hidden passes* approach, the user level code that creates the GUI only has to be run once per frame. It allows for adaptive layouts as the layouting is deferred to the end of the frame, after all widgets have been created. As a tradeoff, the interactions can also only be processed at the end of the frame.

However, with this approach, event handling by widget function return value is not possible. The interactions cannot be handled until later when the frame is finished and the final layout is computed. In that case, input events can be handled by callbacks or by setting user provided variables to inform the user over which interactions occurred once the GUI frame is finalized.

2.7 Styling

The existing IMGUI implementations follow different styling approaches. In Dear ImGui, the current style can be modified stack-based. For example, a call to `PushStyleColor` will change the color for all upcoming widgets, until another style color is pushed onto the stack or until it is removed from the stack via a call to `PopStyleColor` [Cor14]. This enables the application programmer to implement cascading styles. Other style variables like sizes and spacings can be set similarly. Nuklear additionally offers *skinning* to the stack-based style adjustments. It allows using color textures to change the look of widgets [Met15]. In micoui, next to setting style variables manually, the internal `draw_frame` function is implemented as a function pointer [rxi18]. The application programmers can exchange the default implementation at runtime for their own ones to be in full control over what is being input into the draw list for a frame. This will alter the look of all the widgets that are drawn with a frame around them.

An attempt to maximize flexibility would build upon Nuklear's approach of allowing both stack-based style manipulation and skinning as well as the option to override default implementations of primitives similar to microui. This could be done either with function pointers or virtual functions. To be consistent, the overridden primitive functions could also be stack based. This would allow buttons in the furniture planner to be styled differently, depending on the context. While buttons on object-mounted UI could have blue rounded buttons, the UI the users use to create more furniture could have orange hexagonal buttons. In a flexible UI implementation, the application programmers can extend the set of possible styles like the hexagonal layout.

3 Discussion

The same GUIs can be created with both IMGUI and RMGUI systems, but they differ in the complexity of the resulting code. The base of the discussion between both approaches is the differences in their API. One of the characteristics which can be used to classify and compare APIs for libraries is data *retention*. It is concerned with how much state the library retains internally that the application manually has to synchronize. A library's API, which requires explicit updating to keep its internal state in sync with the application's data, is called a *retained mode* API. If no explicit synchronization is necessary because the application communicates the relevant information to the library each time its service is requested, the API is called *immediate mode*. This API difference is not limited to GUIs, but can be seen in other libraries as well.

An example of a library that could be designed both with a retained mode API and with an immediate mode API is a physics engine. For the retained mode version, the application programmer has to be concerned with the lifetime of physics objects like joints. If a joint should be simulated when and as long as the application user presses a button, each frame the state of the input has to be queried. Depending on the button state, it has to be checked

if the joint already exists from the last frame and has to decide if it has to be deleted or created or if no action is required. This results in nested conditionals for a seemingly simple requirement. For an immediate mode physics library, no joint information would persist between frames. Only the button state would be queried each frame and if the button is being pressed, a joint would be simulated this frame. Leading to a reduction in code size and complexity [Mur04].

The feature where the furniture-mounted UI is hidden if the user is too far away would lead to code similar to retained mode physics example. Each iteration, the distance would be checked and depending on the distance, either no action is required, or the UI has to be created or deleted. In contrast, in an IMGUI MR implementation, after checking the distance to the object, either the full UI is shown or the small indication that there is a UI if the user comes closer.

4 Conclusion

It was shown that the 2D IMGUI concept is adaptable to 3D UIs to be used in the domain of MR applications. This topic is worth investigating as current MR GUIs make use of RMGUIs while immediate mode APIs tend to be more flexible in real-time applications or programs that operate in some kind of simulation loop, like games or MR applications. Immediate mode APIs typically result in less and easier code for the application programmer. Each frame or simulation step is treated independently. In an immediate mode GUI library, the creation or deletion of elements does not need to be announced. They exist in one frame and might not exist in the next. If the UI in mixed reality is implemented as game objects that have to be created and deleted as with the mixed reality toolkit [mrt22b], additional effort has to be spent on implementing the UI transitions.

RMGUIs require events to be handled in the form of callbacks or abstracted forms of callbacks like virtual functions. The virtual function approach is similar to the use case of the mixed reality toolkit for Unity, where the 3D widgets can be combined with a specialized parent class where methods will be called in the case of UI events [Mic21b].

While this work presented many new aspects of IMGUI for MR applications, the exact impact is unknown and requires further work. The next steps consist of implementing a prototypical MR IMGUI system. With it, the effect of the different API on the code quality and productivity of the programmers can be quantified more precisely.

References

[Bar05] Sean Barrett. Immediate mode guis. *Game Developer Magazine*, 12:34–36, 2005.

[BFH01] Blaine Bell, Steven Feiner, and Tobias Höllerer. View management for virtual and augmented reality. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 101–110, 2001.

[Cor14] Omar Cornut. Dear imgui. https://github.com/ocornut/imgui, 2014. accessed: 01.04.2022.

[Fus21] Luigi Fusco. Providing a remote debuggui to dpl. Sep 2021.

[Hec16] Jeff Hecht. Optical dreams, virtual reality. *Opt. Photon. News*, 27(6):24–31, Jun 2016.

[Kay96] Alan C Kay. The early history of smalltalk. In *History of programming languages—II*, pages 511–598. 1996.

[Met15] Micha Mettke. Nuklear. https://github.com/Immediate-Mode-UI/Nuklear, 2015. accessed: 01.04.2022.

[Mic21a] Microsoft. *About Messages and Message Queues*, 2021. accessed: 05.04.2022.

[Mic21b] Microsoft. *UX building blocks – Interactable*, 2021. accessed: 30.04.2022.

[mrt22a] Mixed reality – pointers. https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/features/input/pointers?view=mrtkunity-2021-05, 2022. accessed: 30.04.2022.

[mrt22b] Ux building blocks – buttons. https://docs.microsoft.com/de-de/windows/mixed-reality/mrtk-unity/features/ux-building-blocks/button?view=mrtkunity-2021-05, 2022. accessed: 30.04.2022.

[Mur04] Casey Muratori. Designing and evaluating reusable components (2004). https://caseymuratori.com/blog_0024, 2004. accessed: 29.03.2022.

[Mur05] Casey Muratori. Immediate-mode graphical user interfaces (2005). https://caseymuratori.com/blog_0001, 2005. accessed: 12.10.2021.

[OEW19] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. An exploration and experiment tool suite for code to architecture mapping techniques. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, page 26–29, New York, NY, USA, 2019. Association for Computing Machinery.

[PŠ20] Pavel Pokorný and Daniel Ševčík. An application for solving truth functions. In Radek Silhavy, editor, *Intelligent Algorithms in Software Engineering*, pages 341–351, Cham, 2020. Springer International Publishing.

[Roo03] Ton Roosendaal. Blender interface.c api toolkit notes. https://git.blender.org/gitweb/gitweb.cgi/blender.git/, 2003. accessed: 05.04.2022.

[rxi18] rxi. microui. https://github.com/rxi/microui, 2018. accessed: 30.04.2022.

[Uni22] Unity Technologies. *MonoBehaviour.OnGUI()*, 2022. accessed: 30.04.2022.

[vB19] Simon Lennart van Bernem. Nbui. Master's thesis, FH Aachen, 2019.