Makepad Whitepaper

Eddy Bruël
Rik Arends

1. Introduction

In this whitepaper, we introduce Makepad, a novel application framework and IDE for developing VR capable applications. Makepad is developed in Rust, and supports WebAssembly as a compilation target. As a result, Makepad applications are capable of running both natively and in the browser, using WebXR.

1.1 History

As we came out of the 80s, with its separate text editors and compilers and debuggers the 90s introduced the IDE, which means 'Integrated Development Environment'. In this environment, all the tools needed to write, run and debug applications are tightly integrated to enhance productivity. Most notable were Delphi, Visual C++ and Visual Basic. We had visual editors for UI with standard components, debugging, autocomplete, the works. However as the web-application wave of the mid 00's started to take hold, all these technologies were lost to the developer. Instead we had to handwrite applications in HTML and CSS and JS. We have still not remotely equalled the level of sophistication we had in the 90s with web, today.

When we built Cloud9 IDE and its core component, the code editor ACE using HTML and the webstack, the contortions needed to build even basic UI (an editor) became painfully obvious. However, we managed to build it. So far so good for HTML. Unfortunately, it was completely stuck at 'basic code editor', and we didn't have the performance space available to take it much further.

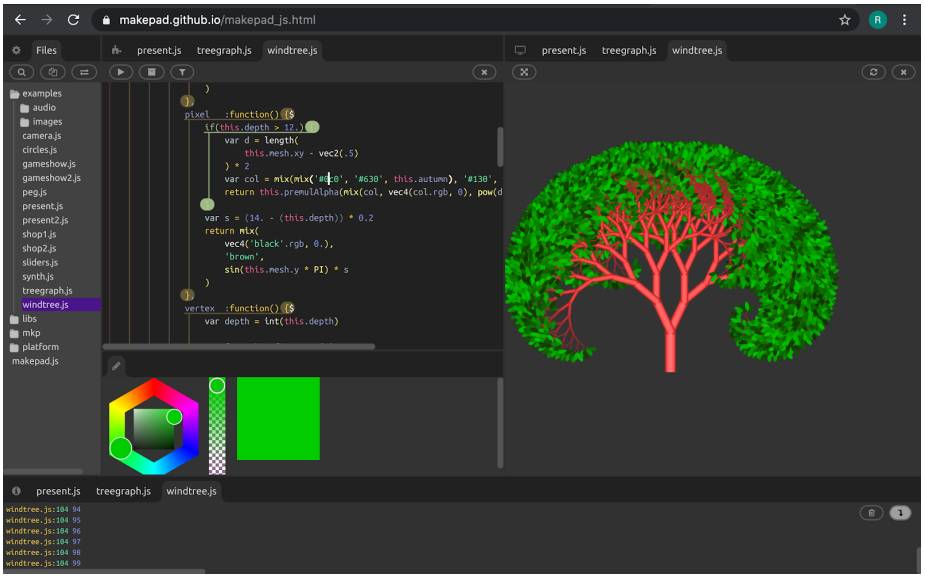
Then Bret Victor made a series of presentations (add links) that showed prototypes of how we could move beyond the basics, if only we could draw more UI and extend the editor with more visual debugging and design tools. With the release of WebGL, a path to reinvent the way we draw applications on web, became feasible. This triggered a decade long search to see if we can do better, and improve fundamentally how we draw a UI on the web, as a necessary dependency to go further than a plain code editor.

CPU clock speeds have plateaued already for many years. This means that for a single thread the CPU hasn't been getting much faster. As chips gets smaller, most of the die space has been going to having more cores, and towards the GPU. For integrated chips the GPU can easily take up half the available space now, or more even. So the key to reinventing how we draw, is to include exposing this programmable resource to the developer. So far with HTML, but also with desktop frameworks, we have been using the GPU only deep down underneath to accelerate things we were already doing (drawing

images, text, etc). Nobody has exposed the programmable fabric of the GPU in the drawing we do to build end user applications, except for game engines.

So, the challenge was to build a new way to make interactive applications in a browser, that fully exposed the GPU to the developer in a way that was as easy as possible, but no easier. The acceptance test of this API is first and foremost to be able to build a developer environment for its own stack, on its own stack. The idea being that if you can't write your own IDE on your own drawing APIs, the APIs are not good enough.

After many experiments we have settled on a model where shaders behave simply as functions, written in the host syntax or near host syntax (JS first, now Rust+GLSL) where passing data from the CPU to the GPU is made as easy as possible, whilst maintaining maximum performance.



* The final JS prototype of makepad, which showed the feasibility of redoing all of the UI using webGL, and the fundamental performance problems of using JS.

An API such as Canvas (web) exposes drawing as a series of high level primitives such as Circle or Rect with only very basic styleable properties (color, outline). Shaders, on the other hand, allow you to style pixel areas using full programs. People have even written DOOM to run entirely on the GPU (shadertoy link). Makepad therefore exposes building your own drawing apis based on the primitives the GPU exposes, whilst maintaining a simplicity very similar to Canvas. By exposing the GPU primitives using an immediate mode composable API.

JavaScript, because of its GC and unpredictable performance, ultimately failed to be usable to replace the entire render stack, and the process stalled for a while in late 2018. Fortunately, however, WebAssembly (Wasm) became available and the possibility to redo the JavaScript prototype using Rust+Wasm became real.

And thus after a year of learning and developing Rust we have a new stack which we present in this whitepaper.

1.2 The Application Framework

The application framework consists of a set of APIs for developing VR capable applications that can run both natively and on the web.

The most important API is the render API. The render API is conceptually similar to Dear ImGui or React. In particular, it is based on the idea of immediate mode. Immediate mode allows you to write code as if every draw command is executed and drawn to the screen immediately (hence the name). This is different from retained mode, where you have to maintain an often complex data structure that represents the scene to be drawn. As an example of such a data structure, consider the DOM in the browser. In fact, frameworks such as React were partially developed to mitigate the difficulty of manipulating the DOM in an efficient way.

An important part of the render API is the shader compiler. The shader compiler is an extension for the Rust compiler that allows the embedding of shader code into ordinary Rust code. This shader code is validated at compile time, and then transpiled at runtime to the shader code expected by the target graphics API (OpenGL, DirectX, or Metal).

The render API also contains platform abstractions that allow applications to run both natively and on the web. In addition to the render API, the framework also provides a collection of builtin widgets to ease the development of applications.

1.3 Makepad as an IDE

The IDE is designed to be the best tool for building applications with the application framework, but the framework itself is not dependent on the IDE. The IDE also serves as a real world example application built using the framework.

The two core features of the IDE are live coding and collaborative editing. Live coding allows the developer to make changes to large parts of the application without having to recompile or restart the application, and significantly reduces restart times in those cases where a restart cannot be avoided. This intention is to significantly reduce overall iteration time when developing new features, thus increasing developer productivity. A live coding system consists of an IDE with a display server and the application being developed. Live coding is discussed in more detail in section 2.

Collaborative editing allows multiple users on different computers to make changes to the same application at the same time. A collaborative editing system consists of a server and multiple clients, each of which has a full copy of the application state. Clients notify the server when they make a change to the application state, and the server notifies the other clients of this change. A concurrency control algorithm is used to keep the server and clients

in sync. In this setup, both the IDE and the application act as ordinary clients. One can thus envision scenarios where multiple users develop the same application using different IDEs on different computers, or where multiple applications cooperate with each other by sharing the same state. Collaborative editing is discussed in more detail in section 3.

1.4 The power of Rust

Why rust was made by Mozilla
Why its better than C++
Typesystem for the win

2. Architectural Overview

2.1 Introduction and Overview

In this section we will go through the main elements of the architecture of the Makepad framework. Section 2.2 briefly explains how to build the IDE from source. Section 2.3 gives an overview of the directory structure of the project. The remaining sections each highlight an important aspect of the application framework. Section 2.4 discusses platform support. Section 2.5 discusses how the immediate mode API works. The actual render architecture is discussed in section 2.6, whereas the use of shaders is discussed in section 2.7. Section 2.8 and 2.9 highlight the important problems of layout and animation using an immediate mode render API, respectively. Finally, section 2.10 touches on how Makepad intends to support VR.

2.2 Building the IDE from source

The following instructions can be used to build a copy of the IDE from source:

- Install Rust
  https://www.rust-lang.org/tools/install
- Clone the Makepad repository:
  `git clone https://github.com/makepad/makepad`
- Change to the Makepad directory:
  `cd makepad`
- Build and run Makepad
  `cargo run -p makepad --release`

2.3 Directory Structure

The Makepad repository is organised into the following directory structure:

examples/
    bare_example/ - desktop example showing bare render api
    widget_example/ - desktop example showing simple widget use
makepad/ - the makepad IDE crate
    app/ - makepad IDE desktop application
    wasm/ - makepad IDE wasm application
    hub/ - makepad filesystem and compile hub
render/ - main platform and rendering crate
    mprstokenizer/ - shader and rust tokenizer
    bind/ - platform bindings
    shader_ast/ - shader ast macro
    tinyserde/ - serde replacement
    vector/ - vector libs (font rendering e.a.)
resources/ - fonts

widgets/ - widget crate

2.4 Platform Support

The Makepad render crate contains several platform abstractions. The following platforms are supported:

- Linux/OpenGL
- OSX/Metal,
- Windows/Dx11
- Wasm/WebGL

The IDE and other applications can be compiled for all these platform without having to change any application code. However, there are some caveats: since Wasm applications don't have access to the native file system, the way the IDE accesses files works differently for the Wasm version. Furthermore, since it is currently not feasible to compile the Rust compiler to Wasm, compiling code is not available in the Wasm version of the IDE.

2.5 Immediate Mode

In immediate mode drawing and UI, displaying anything visual is done with calls that simply state "Draw a text string/rectangle/button/list/etc". This is similar to how the Canvas API in the browser works, as contrasted with SVG, which requires an XML document representing the scene to be drawn.

Having an immediate mode API saves the developer from having to create these data structures, which is especially handy when the things to be drawn are dynamic in response to user input. Frameworks such as React map provide an immediate mode API to the user, but use a retained mode API behind the scenes. In the case of React, the retained mode API is the DOM.

Makepad has what we call a dual-immediate mode API. This means it has an immediate event-flow, and an immediate draw-flow. As compared to dear imgui, which has a single immediate-mode flow, having a separate ‘draw’ and ‘handle event’ flow means you can send many more events independent of the drawflow cheaply, as well as reordering the event-handling to fit an order different than needed for drawing. The immediate mode flows allow makepad to update and handle events in the microsecond range, which is ideal for performance constrained environments.

2.6 Render Architecture

In the immediate mode draw flow, a tree of draw commands is generated. The different back-ends (webGL, metal, etc) then execute these trees of draw commands with shaders.
Draw commands are packed into an ‘instance buffer’, which are then executed all at once as single draw-call per shader type. So you can draw 10.000 rectangles, which as long as they use the same shader only becomes a single draw-call on the backend API. This is the main reason makepad is so fast in a browser, since the UI consists of the minimum amount of

separate draw-calls. All the rest is simply handled in the form of large buffers that don't need to be touched by any JS code.

In order to parameterise the styling done by the shader, we need to pass along data from the CPU to the GPU into these instance buffers. This is the main area where the makepad shader compiler is used, as it dynamically creates the per-instance struct layout for each shader. This way you can trivially declare a variable in a shader, and pass data into it from the immediate mode draw code.

The drawing API in makepad is built directly on top of shaders. Shader programs have two main functions called the 'vertex' shader and the 'pixel or fragment' shader. In makepad these two are called 'pixel' and 'vertex' respectively. The pixel function returns a vec4 color, and the vertex function returns a position.

Since clipping often requires the use of 'stencil' features of the graphics API, and this has a lot of overhead especially in 3D, makepad uses 'vertex shader clipping'. A technique in which the vertex shader simply clips the geometry to a known rectangle. This has limitations, but as long as we draw primitives built out of clip-aligned rectangles it is not a problem.

2.7 Shaders

Shaders in makepad are written using GLSL or Rust-like syntax, and they can use classical single inheritance as a way to quickly reuse and style different base shaders.
The most basic shader we have is the Quad shader, a simple 2 triangle rectangle where the pixel shader is trivially overridable. Since the vertex shader of Quad contains a bit of logic to provide the clipping and scrolling logic, inheriting drawing from the Quad shader is what most UI components do.

In order to provide a more human-friendly API in the pixelshader to draw, we have taken inspiration from Leonard Ritters 2D vector API https://www.shadertoy.com/view/lslXW8
Thus the base shader provides this API as a form of a 'functional 2D shader based canvas API' with primitives as circle, rounded box, etc. This combination of an extremely performant way to instance primitives from the CPU side, and having an easy to program pixel shader based styling system is what sets the makepad APIs apart.

2.8 Layout

If you have an immediate mode drawflow, as we know from the canvas API, doing layout becomes a significant and unsolved problem. Most retained mode API's provide layout via an 'engine' that processes the data structure, and we only know where things will end up after all of the data has been processed. Trying to use knowledge about 'where' something will end up with HTML creates the well known stalls since the layout engine is running in a different thread from the user code and thus it effectively creates a cyclic dependency.

In makepad this is solved using a concept called Turtles. The name borrowed from the original Logo turtle. The idea is that as the immediate mode draw-flow executes, you move a turtle around, also a turtle records the dimensions of what has been drawn. This allows for doing ‘content-sized layouts’ which are extremely common in UI. Turtles form a stack (turtles all the way down), and on the closing edge of the immediate-draw walk they are allowed to ‘move’ what has just been drawn. This allows for features like line breaks and float layouts whilst still using a canvas-like immediate mode API. It also allows for fully nestable layouts. This API has some limitations where ahead-knowledge of the size of neighbours is needed , however they seem adequate for most UI’s and it’s extremely performant, and hopefully easier to understand than most layout engines such as flexbox.

2.9 Animation

Animation in makepad can be done in two ways. The most efficient way is what is often used for mouse-over animation, and it runs on the ‘event’ system. This means you start an animation and you get ‘animation’ events. In these animation events, using a pointer into the generated draw commands called an ‘Area’ values can be directly updated without re-executing the drawflow. This is very cheap as after such an animation event the draw backend simply can re-execute the commands that were generated before but now with updated data. However, since the layout is executed in the drawflow this only offers very limited animation capability. If you want to animate something that pushes other things around in the layout, we have to run an animation on the draw-flow. This works by simply invalidating the view area in the drawflow until you wish to stop animating. This kind of animation is also often using basic exponential counters such as ‘while t>0 t = t * 0.9, redraw’. The code folding and tree animations in makepad are of the draw flow animation kind, whilst the mouseovers of buttons and the items in the tree are of the animation event kind.

2.10 VR and WebXR

Makepads rendering is fully 3D ready, since all it requires to render in 3D is a different view matrix. Z-buffering is by default turned on, even in 2D and UI’s can take up 3D space or even be drawn using 3D geometry.

Makepad will compile to a native Oculus Quest application using the Rust/ARM toolchain, as well as run on webXR.

For executing on webXR the WASM blob sends draw-command trees to JS, which are executed there. The WASM blob is treated as a simple single entry point message-passing mechanism. The JS / browser sends all input events to WASM/Rust, and WASM/Rust returns simply a list of draw commands to be executed by JS on webGL.

For playing with user programs in webXR, this draw-command tree in JS is turned into a render-server, so that a user program running in a worker can simply send over these draw command trees. This makes the IDE and VR environment independent of possible infinite loops in the user program. This will facilitate playing with the live-coding capabilities of

makepad on web, such as editing models or shaders. However what remains a challenge will be to compile new Rust to WASM since that requires the full Rust compiler stack on a native platform. Hotloading new Wasm continuously is also unlikely to be stable, or fast enough using the browser stack. For these cases makepad will run natively without the overhead and stability problems of the browser. However, after working on a program natively, it should still compile and be deliverable on webXR.

3. Live Coding

3.1 Introduction and Overview

It is well known that developer time is expensive. For many software companies, developers are the biggest cost item on the budget. Consequently, there is a lot of interest in finding ways to improve developer productivity.

One of the main obstacles towards developer productivity is long iteration times. Iteration time is understood as the time between the developer making a change to the application, and the them seeing the results of that change. Long iteration times force the developer to either sit by idly while they wait for the results to come in, or context switch so they can work on other tasks in the meantime. Unfortunately, context switching also takes a long time, and is thus in itself an obstacle towards developer productivity.

Overall iteration time can be seen as the product of the number of required iterations to make the desired change to the application, and the average iteration time. We identify two primary sources of long iteration times: compile time and debug time. Compile time is obvious: the longer an application takes to recompile after the developer makes a change, the longer it takes for the them to see the results of that change. Compile time thus affects the average iteration time. Debug time, on the other hand, is less obvious, and therefore often overlooked. When the developer makes a change to the application, and the result of that change is not what they expect, the developer needs to spend time identifying the cause of the erroneous behavior. Since a typical debug session involves multiple compile cycles, debug time affects the number of required iterations.

Programming languages can be classified as being statically or dynamically compiled. Statically compiled languages are fully compiled before the resulting program is executed. Examples of statically compiled languages are Rust and C++. Dynamically compiled languages, on the other hand, are compiled on the fly, using just-in-time compilation, while the program is being executed. An example of a dynamically compiled language is JavaScript.

Statically compiled languages typically have higher compile times, since they have to do all the compilation work up front. However, doing so has the advantage that the compiler can perform many checks that would otherwise have had to be postponed until run-time. In particular, statically compiled languages often feature strong type systems, which helps to detect a large category of bugs known as logic bugs. In addition, languages such as Rust feature a borrow checker, which helps to detect yet another large category of bugs known as memory safety bugs. Since bugs are usually much easier to find at compile-time than at run-time, statically compiled languages typically have lower debug times.

Dynamically compiled languages, in contrast, typically have lower compile times, since they can amortize compilation time over the execution of the program. However, since a

just-in-time compiler runs while the program is executed, it must keep turnover time as low as possible. Not doing so would lead to noticeable hiccups. Consequently, just-in-time compilers cannot perform as many checks as ordinary compilers, so these checks have to be postponed until run-time. As we already pointed out, bugs are usually much easier to find at compile-time than at run-time. As a result, dynamically compiled languages typically have higher debug times.

This leaves us with something of a conundrum. We can either use a statically compiled language such as Rust, which has lower debug times, but higher compile times, or a dynamically compiled language such as Javascript, which has lower compile times, but higher debug times. To bring down overall iteration time, we either have to reduce the compile time of statically compiled languages, or the debug time of dynamically compiled languages. Unfortunately, both of these are currently unsolved problems.

Reducing the compilation time of statically compiled languages is a very hard problem, that compiler development teams over the world have been working on for many years. The inherent difficulty is that the compiler needs to perform many checks before compilation is finished, and these checks take time. In theory, it is possible to omit some of these checks. For instance, unlike Rust, C++ does not have a borrow checker. Instead, it simply considers memory safety bugs undefined behavior, meaning that although the program compiles just fine, its subsequent behavior is undefined. By allowing for undefined behavior in much more cases, the C++ compiler should in theory be able run faster than the Rust compiler. Unfortunately, this just shifts the burden of finding the resulting bugs to the programmer at run-time, where they are much harder to find, and thus doesn't meaningfully reduce overall iteration time.

Similarly, there have been many attempts to reduce the debug time of dynamically compiled languages. For instance, TypeScript is an attempt to add static type checking to Javascript by extending the syntax of the language. This helps the developer to catch a large amount of logic errors before the program is ever executed, leading to more robust programs, and thus lower debug times. Unfortunately, TypeScript requires a compilation step to perform type checking, and convert itself to plain JavaScript. This effectively turns JavaScript into a statically compiled language, which loses most of the benefits of dynamic compilation such as low compile times. In other words, we traded debug time for compile time, but didn't meaningfully reduce overall iteration time.

To find a way out of this dilemma, Makepad proposes a different approach: since we cannot improve the debug time of a dynamically compiled language without turning it into a statically compiled language, anyway, our starting point is to use a statically compiled language. The primary source of iteration time in a statically compiled language is compile time. We've already seen that reducing the compilation time of statically compiled languages is a very hard problem. However, those attempts primarily focused on reducing the average compilation time. Recall that overall iteration time is the product of the number of required iterations and the average iteration time. What if we could reduce the number of required compilations instead?

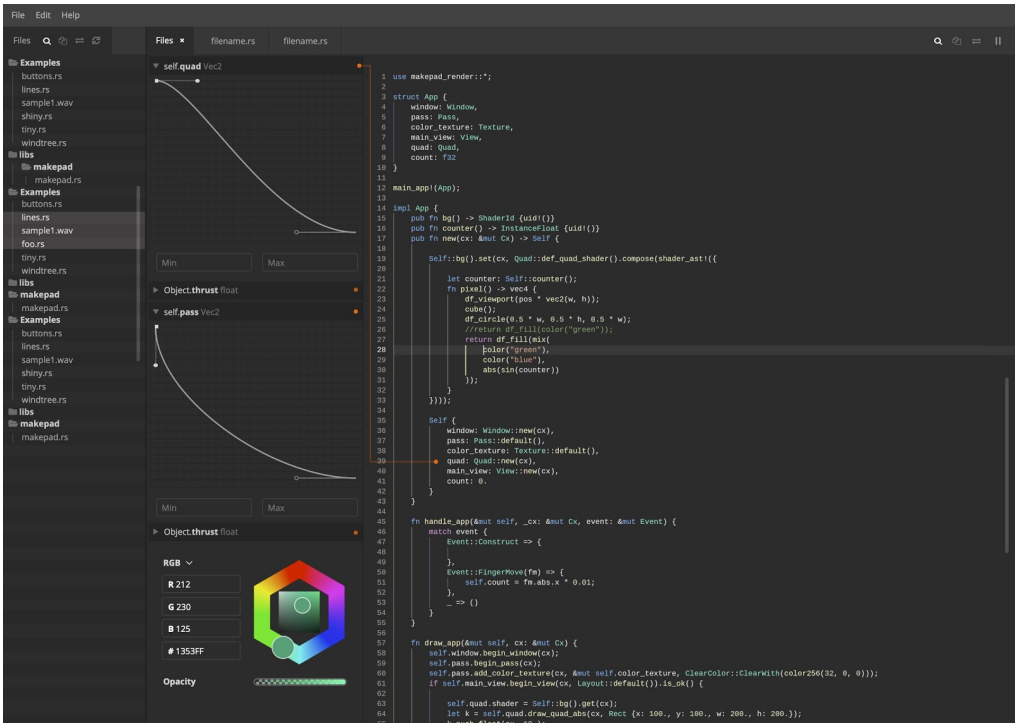
Our proposed approach is based on the assertion that most changes to a programming don't actually require changing the code. Rather, these changes can be made by only changing the data instead. Examples are changing the layout and/or style of a widget in an application, changing the animation of a character in a game, etc. In these cases, a compilation step is not required. Our approach, which we call live coding, is thus a hybrid system, in which changes to the code still trigger a compilation step, but changes to the data can be made at run-time. Compilation time is still as high as before, but since changes to the data no longer trigger a compilation step, the number of required compilation steps is reduced. As a result, the overall iteration time is reduced as well.

In the remainder of this section, we will describe live coding in more detail. Section 2.2 focuses on what live coding is, and section 2.3 on how it is implemented.

3.2 What is Live Coding

The goal of live coding is to minimize the overall iteration time between making a change to an application and seeing the result of that change. The primary idea behind live coding is that although average compilation time cannot be reduced, the number of required compilations can.

A live coding system consists of an application and an IDE. A prototype of this IDE can be seen in the image below:



The IDE consists of a code editor to make changes to the code, and other widgets to make changes to the data of the application. Examples of such widgets include:

- A color picker
- A bezier shape editor
- An animation timeline

As explained before, changes to the main application code code still require a compilation step. This is not fundamentally different from other IDEs. However, changes to the data can be made while the application is running, using such widgets as mentioned above. This is the live aspect of live coding: the live refers to the fact that the application can stay alive while the changes are made.

In our discussion of code versus data so far, our definition of data included code written in domain specific languages (DSL). This justifies the term live coding, rather than just live editing. Unlike the main application code, which is compiled at compile-time by the IDE, code written in these DSLs are either compiled or interpreted at run-time by the application. This greatly expands the range of things that can be expressed as data, thus further reducing the number of compilation steps required to make changes to the application. An example of such a DSL that is of particular importance for Makepad is shader code. Makepad's UI render system is heavily based on shaders. By making shader code a DSL, a large amount of styling can be done using live coding.

To minimize friction between code and data, data or DSL code that can be live coded is embedded in the main application code, using literals such as numbers or strings, and specially marked as such using a macro. These specially marked parts of the code are recognised by the IDE, allowing the latter do display a dedicated widget for editing the corresponding data. Any changes made to this data in such a widget is instantaneously reflected in the code. Conversely, any changes made to this data by editing the code directly are instantaneously reflected in the corresponding widgets.

In some cases, a compilation step is still unavoidable, in which case the application needs to be restarted. To minimise the disruption caused by such a restart, we would at least like the application to restart in the same state that it was before. This is a very difficult problem to solve in general, but can at least be partially solved by sharing the application state between the application and a server. Any changes made to the model by the application are forwarded to the server, so it can update its copy of the application state accordingly. Once the application restarts, it can obtain the latest copy of the model from the server.

Another aspect of live coding is that applications do not render themselves directly. Instead, they send a stream of render commands to a the IDE, which also acts as a display server, in a similar vein as X11/Wayland. This allows multiple applications to share the same render context. This is particularly important in VR environments, where typically only one render context is available. As an additional advantage, having a dedicated display server allows us to further reduce application restart times, because many resources used by the application can be cached in the IDE, rather than regenerated each time the client restarts.

To summarize, the key ideas behind live coding are:

- Avoiding recompilation whenever possible, by making it possible to change the application state while the application is running
- When recompilation cannot be avoided, minimize restart time, by caching the application and render state on a separate server.

There are many advantages to live coding. Among other things, it allows for much faster exploration and tweaking of the creative design space, and much tighter integration of the developer and designer workflows. Also not unimportant is the fact that it is simply more fun to develop software with short iteration cycles. All these factors are expected to significantly increase developer productivity, which is the goal we set out to achieve.

3.3 How is Live Coding Implemented

The implementation language of choice for Makepad is Rust. Rust is a statically compiled, multi-paradigm programming language developed by Mozilla. This fits with our approach of starting with a statically compiled language, and then trying to reduce the number of required compilations to reduce overall iteration time. Moreover, Rust focuses on performance and safety, in particular memory safety. This focus on performance is particularly important for Makepad, which is intended to run on VR systems, where consistent frame rates are extremely important. The same performance would have been obtainable by C++, but our experience with Rust has shown that its focus on safety leads to programs that are both most robust and easier to debug. In other words, the use of Rust by itself reduces our own iteration time, at least compared to C++.

Another advantage of Rust is its relatively strong tooling for WebAssembly, when compared to C++. WebAssembly is an open standard defining a portable binary code format for executable programs. Such a portable binary code format comes with many advantages. At the time of this writing, all major browsers except Internet Explorer are capable of running WebAssembly programs. This enables one to write applications that, with some care, are capable of running natively as well as in a browser. Since WebAssembly is an open standard, one could conceivably even write their own WebAssembly run-time, and use that as an application framework in a similar vein as Electron. With these use cases in mind, Makepad has been designed from day one supporting WebAssembly as a compilation target.

In order to implement live coding, Makepad leverages Rust macros to identify data or DSL code that can be live coded. Recall that such data or DSL code is embedded in the main application code. As a very simple example, consider the following code: `pick!("red")`. This identifies a live codeable color, specified as a string, by wrapping it in a call to a macro called `pick` (Rust identifies macro calls by appending a ! to them). The IDE is capable of detecting such macro calls, and thus show a color-picker instead of a text-editor to change the color.

This in itself is not new. Many HTML/CSS editors also have this feature. However, Makepad takes this approach one step further by using a collaborative editing system, consisting of a collaboration server and one or more clients. The application state is distributed between the server and the clients. When a client makes a change to the application state, it notifies the server, which in turn notifies all the other clients. A concurrency control algorithm is used to

keep the server and the clients in sync with each other. Within this scheme, the application and the IDE are just ordinary clients. This allows the application state to be shared between the application and the IDE, so that changes made in the one are reflected in the other, and vice versa. Moreover, it allows the application to recover its state after it is restarted due to a recompilation or crash. Collaborative editing is discussed in much more detail in section 3.

The render stack in Makepad uses what we call deferred rendering with a virtual immediate mode API. Deferred rendering, in this context, means that render calls are recorded in a command buffer, to be executed at a later time, rather than executed immediately. In contrast, immediate mode is a style of API design in graphics libraries where calls made by the client cause graphical objects to be rendered directly to the display. Makepad's render API is immediate mode, in the sense that applications behave as if objects are rendered directly to the screen. However, the API is virtual in the sense that under the hood, immediate calls are instead translated to deferred calls. In other words, although Makepad's render API behaves as an immediate mode API, it really just records render calls under the hood. Bridging this gap between immediate and deferred rendering is one of the major hallmarks of Makepad's API design, and what makes Makepad applications so easy to write..

Because Makepad uses deferred rendering, the rendering of applications can be decoupled from the applications themselves: rather than render themselves directly, Makepad applications send streams of render commands to a the IDE, which also acts as a central display server. The IDE uses a display protocol in a similar vein as X11/Wayland (albeit it much more lightweight). Having a centralized display server has several advantages. For instance, VR environments typically have only one render context. To be able to run multiple applications within the same VR environment, they thus have to share the same render context. A centralized display server is the most obvious solution to this problem. Another advantage is that graphical resources used by the application, such as textures and shaders, are owned by the IDE rather than the application itself. Consequently, if the application has to restart, these resources can be reused, rather than regenerated, which reduces restart time.

Because the entire application state is ultimately represented as code, we get a number of benefits. For instance, since all changes to the application state are implemented as changes to the code, undo/redo functionality can be implemented in a similar way as in ordinary code editors, where this is a well understood problem. Moreover, only allowing textual changes allows us to use a variant of operational transform as our concurrency control algorithm of choice for collaborative editing. Operational transform, although not trivial to implement, is at least relatively well understood. In contrast, implementing concurrency control for multiple different data types, where one type of data can be embedded in the other, is a significantly harder proposition.

4. Collaborative Editing

4.1 Introduction and Overview

One of the core features of Makepad is collaborative editing. Collaborative editing is the ability for multiple users on different computers to simultaneously edit the same document. The fundamental problem in collaborative editing is this: given multiple concurrent changes to a shared document, how do we ensure consistent results between the different clients? Solving this problem is the job of concurrency control algorithms.

Concurrency~~Concurrency~~ control algorithms can be classified into being pessimistic or optimistic. Pessimistic algorithms require clients to communicate with either a central coordinator or other clients before they are allowed to apply a change locally. Optimistic algorithms, on the other hand, allow clients to apply a change locally before informing the other clients of the change. If multiple clients edit the document simultaneously, a conflict resolution algorithm is used to bring all clients into the same final state.

Pessimistic algorithms are easier to implement than optimistic ones: because all clients have to agree on a change before it can be applied locally, no conflicts can occur, and therefore no conflict resolution is required. However, optimistic algorithms offer a better user experience than pessimistic algorithms when network latency is high, since the result of user actions can be displayed immediately, without having to wait for a response from either a central coordinator or other clients.

Concurrency control algorithms can be further classified into being centralized, distributed, or decentralized. Centralized algorithms involve a central coordinator with which all clients communicate. Distributed algorithms, on the other hand, have no central coordinators, so the clients have to communicate with each other in order to coordinate their efforts. Decentralized algorithms take a middle road between being fully centralized and fully distributed, by allowing for multiple coordinators. Each client typically communicates with a single coordinator, and the coordinators themselves communicate with either a supercoordinator or each other.

Centralized algorithms are easier to implement than either distributed or decentralized ones: because the coordinator can store an authorative copy of the document and impose a global order on any changes to that document, no special algorithms are required to achieve global consistency. In contrast, distributed algorithms in particular are notoriously difficult to implement. However, they offer better reliability than centralized algorithms in the presence of network failures, since there is no single point of failure, and perform better than centralized algorithms when the network is under heavy load, since there is no single route of communication.

Decentralized algorithms combine most of the advantages of centralized and distributed algorithms: they are harder to implement than centralized algorithms, but much easier than

distributed ones, they are more robust than centralized algorithms in the face of network failures, and they perform better than centralized algorithms when the network is under heavy load. This often makes decentralized algorithms an acceptable compromise in practice.

An example of a decentralized, pessimistic algorithm is used in Croquet, which is an SDK for developing collaborative virtual word applications. Croquet implements concurrency control using a central coordinator, called a reflector. Clients make changes to a shared document, called the model, by sending change messages to the reflector. The reflector broadcasts/reflect these changes to all the clients. The order in which messages are reflected bt the reflector imposes a global order for these messages. In addition, a reflector can store an authorative copy of the model, which consists of a snapshot, plus any changes made to the model since the last snapshot. When the network is under heavy load, clients can transition from one reflector to another that has better response time. To coordinate this transition, reflectors communicate with yet another reflector that sits above the others.

Croquet's approach is well suited for collaborative virtual word applications, where some round-trip delay is usually acceptable. Its decentralized nature helps to keep this delay to a minimum. New network technologies, such as 5G, are expected to bring round-trip delay down even further. The only area where round-trip delay is still a potential issue is collaborative text editing. The average typing speed is about 200 characters per minute, which amounts to 300ms per character. For professional typists such as software developers, this speed is even higher. Any network delays will thus immediately be noticeable as lag in the user interface, which makes for a very poor user experience. This is particularly a problem when the network is heavily congested, which is often the case at conferences, etc. Being able to transit to another reflector does not help in this case, since the problem is on the client's side.

Since Makepad focuses on being an IDE for developing virtual world applications, collaborative text editing is a particular concern. With this in mind, we propose to use a decentralized, optimistic algorithm for concurrency control. The primary difference between such an algorithm and the one used by Croquet is that changes are immediately applied locally, before receiving a response from the reflector. Since this allows multiple clients to make a change to the same part of the document at the same time, some sort of conflict resolution algorithm is required.

In the remainder of this section, we will describe the concurrency control algorithm we propose to use for Makepad in more detail. The algorithm itself is heavily based on the paper "High-latency, low-bandwidth windowing in the Jupiter collaboration system" by David Nichols, Pavel Curtis, Michael Dixon, and John Lamping. This same paper was also used as the basis for Google Wave's Operational Transform. For the sake of exposition, we will describe the algorithm for the case where there is a single client. In the final notes, we will explain how this algorithm can be extended to cover the case where there are multiple clients.

In section 3.2, we introduce the terminology and notation we will use in our description of the algorithm. In section 3.3, we explain how the client and server can become desynchronised when they both edit the same document, and how resynchronisation can be achieved, provided the client and server do not edit the document at the same time. In section 3.4, we explain how our resynchronisation algorithm breaks down when the client and server edit the document at the same time, and how the conflicting deltas that are created as a result can be resolved by transforming them, provided both deltas were generated when the client and server where in the same state. In section 3.5, we explain how our conflict resolution algorithm breaks down when the conflicting deltas were generates from different states, and how these conflicts can be resolved as well. In section 3.6, we use the synchronization algorithm described in section 3.3 and the conflict resolution algorithm and the conflict resolution algorithm described in sections 3.4 and 3.5 concurrency control algorithm between a single client and the server in its general form. Finally, in section 3.7, we offer some final notes.

4.2 Terminology and Notation

A **state** represents the state of the document on either the client or the server. We denote a state as a pair (x,y), where x is the number of changes from the client, and y the number of changes from the server that have been processed so far. When the client and server are in the same state (x,y), they are said to be **synchronized**.

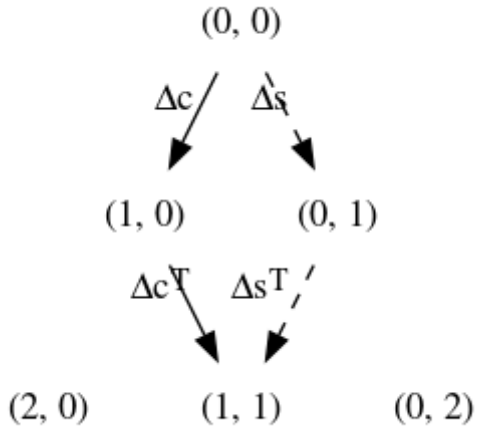
A **delta** represents a change to the document. We denote a delta as a transition function $\Delta : (x_1,y_1) \mapsto (x_2,y_2)$, where $x_1 \le x_2$ and $y_1 \le y_2$. We say that (x_1,y_1) is the **source state** of Δ and (x_2,y_2) the **target state** of Δ. Deltas that originate from the client are denoted Δc, and deltas that originate from the server are denoted Δs. To transition from its current state to another state, the client or server must apply a delta to its current state. The application of a delta $\Delta : (x_1,y_1) \mapsto (x_2,y_2)$ to a state (x_1,y_1) is denoted $\Delta(x_1,y_1) = (x_2,y_2)$.

If the client is in state (x,y) and wants to make a change to the document, it must generate a delta $\Delta_c : (x,y) \mapsto (x+1,y)$ and apply it to its current state. This causes the client to transition to state $(x+1,y)$. The $x+1$ reflects the fact that we have now processed one more change from the client. Similarly, if the server is in state (x,y) and wants to make a change to the document, it must generate a delta $\Delta_s : (x,y) \mapsto (x,y+1)$ and apply it to its current state to transition to state $(x,y+1)$. The $y+1$ reflects the fact that we have now processed one more change from the server.

Deltas are partial functions. The delta $\Delta : (x_1,y_1) \mapsto (x_2,y_2)$ can only be applied to the state (x_1,y_1). This reflects the fact that a change to the document is specified relative to the current state of the document. For instance, a delta that insert a line into the document only makes sense as long as the location of that line is not changed by the application of earlier deltas.

A **transformation function** is a function that takes one or more deltas as input, and returns one or more deltas as output. We denote a transformation function as $T(\Delta_1, \Delta_2, \ldots) = (\Delta_1{}^T, \Delta_2{}^T, \ldots)$. A delta Δ that has been transformed by a transformation function T is denoted Δ^T. If Δ has been transformed by T multiple times, we denote this as Δ^{T^n}, where n is the number of times that T has been applied.

To aid our description of the algorithm, we will sometimes use state diagrams such as the one here below to visualize both the states visited and transitions made by the client and server. Transitions made by the client are denoted with a solid line and transitions made by the server with a dashed line:



4.3 Two-way Synchronization

Consider the following sequence of events:

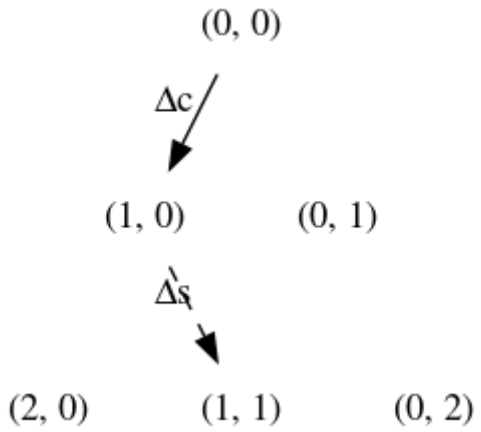
The client and server start in the same initial state $(0,0)$. The client makes a change to the document by generating and applying $\Delta c : (0,0) \mapsto (1,0)$ to transition to state $(1,0)$. Subsequently, the server also makes a change to the document by generating and applying $\Delta s : (0,0) \mapsto (0,1)$ to transition to state $(0,1)$. At this point, the client and server are in different states, and have thus become desynchronized.

Our goal is to resynchronise the client and server so that they are in the same state again. To accomplish this, we introduce the following rule: Whenever the client applies c, it must also send a copy of c to the server, so the server can apply c as well. Similarly, whenever the server applies a delta s, it must also sends a copy of s to the client, so the client can apply s as well.

With this rule in place, let's consider the previous sequence of events again:

The client and server again start in the same initial state $(0,0)$, and the client again makes a change to the document by generating and applying $\Delta c : (0,0) \mapsto (1,0)$ to transition to state

$(1,0)$. However, this time the client also sends a copy of c to the server. When the server receives c, it applies c to transition to state $(1,0)$. Subsequently, the server also again makes a change to the document. However, this time, it does so while in state $(1,0)$ rather than state $(0,0)$. The server therefore generates and applies $\Delta s : (1,0) \mapsto (1,1)$, to transition to state $(1,1)$. The server also sends a copy of Δs to the client. When the client receives Δs, it applies Δs to transition to state $(1,1)$. The entire sequence of events is summarized by the following diagram:



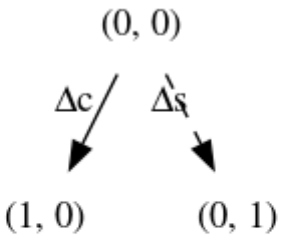
At this point, the client and server are in the same state again, and are thus synchronised.

4.4 Simple Conflict Resolution

The synchronization strategy we described in the previous section only works under the assumption that neither the client nor the server makes any changes to the document while there are still pending changes from its counterpart. To see how our synchronisation strategy breaks down when this is not the case, consider the following sequence of events:

The client and server start in the same initial state $(0,0)$. The client generates and applies $\Delta c : (0,0) \mapsto (1,0)$ to transition to state $(1,0)$, and sends a copy of c to the server. Simultaneously, before receiving c, the server generates and applies $\Delta s : (0,0) \mapsto (0,1)$ to transition to state $(0,1)$, and sends a copy of s to the client.

When the client eventually receives Δs, it is in state $(1,0)$, so it cannot apply Δs to transition to state $(1,1)$. Similarly, when the server eventually receives Δc, it is in state $(0,1)$, so it cannot apply Δc to transition to state $(1,1)$. This state of affairs is summarized by the following diagram:



(2, 0) (1, 1) (0, 2)

What we need is some way to transform Δc and Δs into two new deltas Δc^T and Δs^T, with the property that applying Δs^T after Δc has the same effect as applying Δc^T after Δs.

To get an idea of what such a transformation entails, consider the following example: assume Δc inserts a new line into the document at line 10, while Δs removes a line from the document at line 20. After transformation, Δc^T still inserts a new line into the document at line 10, but Δs^T now removes a line from the document at line 21. This reflects the fact that inserting a new line into the document at line 10 changes the location of the line to be removed to line 21, but removing a line from the document at line 20 does not change the location where the new line is to be inserted.

Formally, what we need is a transformation function T with the following properties:

$$T(\Delta c, \Delta s) = (\Delta c^T, \Delta s^T)$$
$$\Delta c : (x, y) \mapsto (x + 1, y)$$
$$\Delta s : (x, y) \mapsto (x, y + 1)$$
$$\Delta c^T : (x, y + 1) \mapsto (x + 1, y + 1)$$
$$\Delta s^T : (x + 1, y) \mapsto (x + 1, y + 1)$$
$$\Delta c \circ \Delta s^T = \Delta s \circ \Delta c^T$$

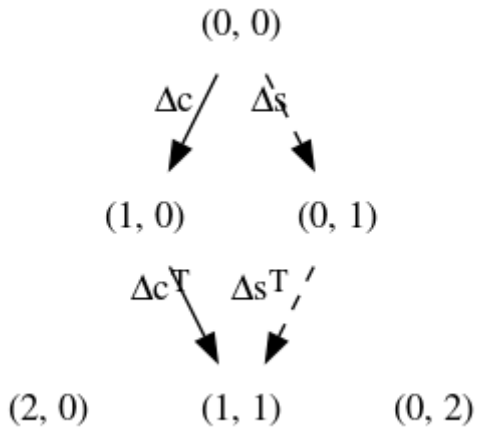
T is only defined when Δc and Δs have the same source state. It takes these two deltas and transforms them into two new deltas Δc^T and Δs^T, with the property that applying Δs^T after Δc has the same effect as applying Δc^T after Δs. To give a full specification of T would take up a lot of space, while not adding much insight into the overall algorithm. We will therefore suffice by assuming its existence.

To simplify our notation in what follows, we introduce two additional helper functions Tc and Ts with the following properties:

$$Tc(\Delta c, \Delta s) = \Delta c^T$$
$$Ts(\Delta c, \Delta s) = \Delta s^T$$
$$(\Delta c^T, \Delta s^T) = T(\Delta c, \Delta s)$$

With T now in hand, let's consider the previous sequence of events again:

When the client eventually receives Δs, it is in state $(1,0)$, so it cannot apply Δs to transition to state $(1,1)$. The client therefore calls $Ts(\Delta c, \Delta s)$ to obtain $\Delta s^T : (1,0) \mapsto (1,1)$, and then applies Δs^T to transition to state $(1,1)$. Similarly, when the server eventually receives Δc, it is in state $(0,1)$, so it cannot apply Δc to transition to state $(1,1)$. The server therefore calls $Tc(\Delta c, \Delta s)$ to obtain $\Delta c^T : (0,1) \mapsto (1,1)$, and then applies Δc^T to transition to state $(1,1)$. The entire sequence of events is summarized in the following diagram:



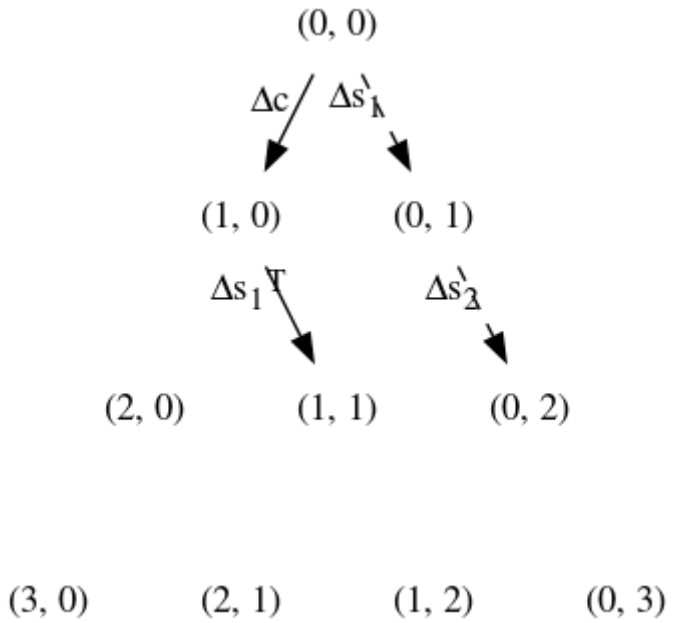
At this point, the client and server are in the same state, and are thus synchronised.

4.5 Complex Conflict Resolution

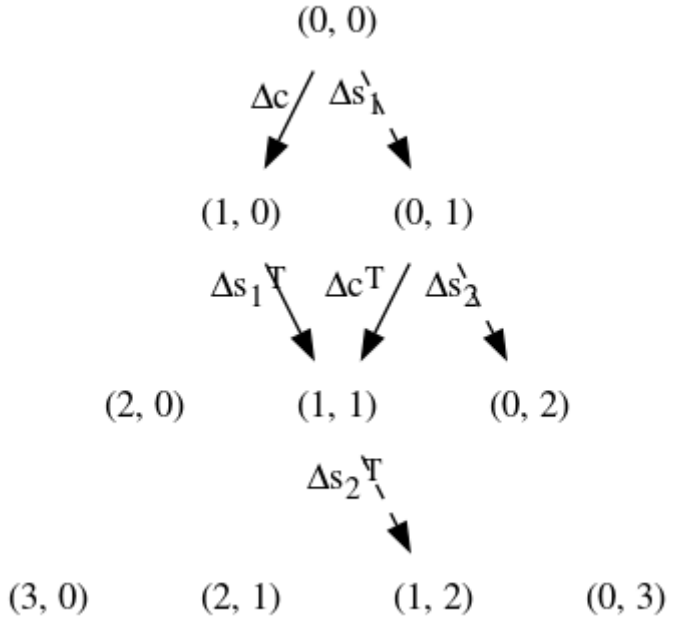
The simple conflict resolution strategy we described in the previous section only works under the assumption that conflicts are simple. That is, the conflicting Δc and Δs must have the same source state, otherwise T cannot be applied. To see how our conflict resolution strategy breaks down when this is not the case, consider the following scenario:

The client and server started out in the same state $(0,0)$. The client has generated and applied $\Delta c : (0,0) \mapsto (1,0)$ to transition to state $(1,0)$, and sent a copy of Δc to the server. Simultaneously, before receiving Δc, the server has generated and applied $\Delta s_1 : (0,0) \mapsto (0,1)$ to transition to state $(0,1)$, and sent a copy of Δs_1 to the client. The client has received Δs_1, called $Ts(\Delta c, \Delta s_1)$ to obtain $\Delta {s_1}^T : (1,0) \mapsto (1,1)$, and applied $\Delta {s_1}^T$ to transition to state $(1,1)$.

At this point, still before receiving Δc, the server generates and applies a second $\Delta s_2 : (0,1) \mapsto (0,2)$ to transition to state $(0,2)$, and sends a copy of Δs_2 to the client. When the client eventually receives Δs_2, it is in state $(1,1)$, so it cannot apply Δs_2 to transition to state $(1,2)$. This state of affairs is summarized in the following diagram:



At this point, the client needs to call $Ts(\Delta?, \Delta s_2)$ to obtain $\Delta {s_2}^T$, but it is not immediately obvious what the value of $\Delta?$ should be. We cannot use Δc, since Δc and Δs_1 do not have the same source state. What we need is $\Delta? : (0,1) \mapsto (1,1)$, so $Ts(\Delta, \Delta s_2) = \Delta {s_2}^T : (1,1) \mapsto (1,2)$. Taking a look at the previous diagram, we see that $\Delta c^T = Tc(\Delta c, \Delta s_1)$ has the required properties. The client therefore calls $Ts(\Delta c^T, \Delta s_2)$ to obtain $\Delta {s_2}^T$, and then applies $\Delta {s_2}^T$ to transition to state $(1,2)$. The entire sequence of events is summarized in the following diagram:



4.6 Two-way Concurrency Control

We are now in a position to describe the concurrency control algorithm between a single client and the server in its general form. We will describe the algorithm from the perspective of the client, but the perspective of the server is similar:

The server was last known to be in state (x,y). Since then, the client has generated and applied k deltas $\Delta c_1, \Delta c_2, \ldots, \Delta c_k$, to transition to state $(x+k,y)$, and sent copies of $\Delta c_1, \Delta c_2, \ldots, \Delta c_k$ to the server. The client has also stored local copies of $\Delta c_1, \Delta c_2, \ldots, \Delta c_k$, until it can confirm that the server has processed these deltas.

The client now receives a delta $\Delta s : (x+i,y) \mapsto (x+i,y+1)$ from the server, where $0 \le i \le k$ is the number of deltas from the client that the server has processed so far. To process, Δs, the client needs to obtain $\Delta s^T : (x+k,y) \mapsto (x+i,y+1)$, and then apply Δs^T to transition to state $(x+k,y+1)$. To obtain Δs^T, the client successively transforms Δs against all $\Delta c_{i+1}, \Delta c_{i+2}, \ldots \Delta c_k$ that have not been processed by the server so far. This leads to the following sequence of transformations:

$$Ts(\Delta c_{i+1}, \Delta s) = \Delta s^{T^1} : (x+i+1,y) \mapsto (x+i+1,y+1)$$
$$Ts(\Delta c_{i+2}, \Delta s^{T^1}) = \Delta s^{T^2} : (x+i+2,y) \mapsto (x+i+2,y+1)$$
$$\ldots$$
$$Ts(\Delta c_k, \Delta s^{T^{k-i-1}}) = \Delta s^{T^{k-i}} : (x+k,y) \mapsto (x+k,y+1)$$

The last transformation gives us $\Delta s^T = \Delta s^{T^{k-i}} : (x+k,y) \mapsto (x+k,y+1)$. The client now applies Δs^T to transition to state $(x+k,y+1)$. Once the server processes the remaining $\Delta c_{i+1}, \Delta c_{i+2}, \ldots, \Delta c_k$ from the client, it also transitions to state $(x+k,y+1)$. At this point, the client and server will be in the same state $(x+k,y+1)$, and will thus be synchronized.

After transitioning to state $(x+k,y+1)$, the client also needs to update $\Delta c_{i+1}, \Delta c_{i+2}, \ldots, \Delta c_k$. $\Delta c_1, \Delta c_2, \ldots, \Delta c_i$ are no longer required, and can be discarded. To update $\Delta c_{i+1}, \Delta c_{i+2}, \ldots, \Delta c_k$, the server transforms each $\Delta c_{i+1}, \Delta c_{i+2}, \ldots, \Delta c_k$ against Δs. This leads to the following set of transformations:

$$Tc(\Delta c_{i+1}, \Delta s) = \Delta c_{i+1}{}^T : (x+i,y+1) \mapsto (x+i+1,y+2)$$
$$Tc(\Delta c_{i+2}, \Delta s) = \Delta c_{i+2}{}^T : (x+i+1,y+1) \mapsto (x+i+2,y+2)$$
$$\ldots$$
$$Tc(\Delta c_k, \Delta s) = \Delta c_k{}^T : (x+k-1,y+1) \mapsto (x+k,y+2)$$

After replacing $\Delta c_{i+1}, \Delta c_{i+2}, \ldots, \Delta c_k$ with $\Delta c_{i+1}{}^T, \Delta c_{i+2}{}^T, \ldots, \Delta c_k{}^T$, the client is ready to process the next message from the server. The server was now known to last be in state $(x+i,y+1)$. Since then, the client has generated and applied $k-i$ deltas $\Delta c_{i+1}, \Delta c_{i+2}, \ldots, \Delta c_k$, to transition to state $(x+k,y+1)$, and sent copies of $\Delta c_{i+1}, \Delta c_{i+2}, \ldots, \Delta c_k$ to the server. The client has also stored local copies of $\Delta c_{i+1}, \Delta c_{i+2}, \ldots, \Delta c_{i+k}$, until it can confirm that the server has processed these deltas.

4.7 Final Notes

The concurrency control algorithm we described in the previous section is only defined for a single client, but can easily be extended to multiple clients, as follows: whenever the server receives a delta from one client, it acts as if had generated the delta itself, applies it locally, and then sends a copy of the delta to all the clients except the one it received the original delta from. Consequently, to all the clients except the one that made the change, it will look as if the server was the one that made the change. This does not affect the transitions made by any of the clients, so synchronisation is preserved..

Each client needs to store local copies of any changes it makes to the document until it receives confirmation from the server that it has processed these changes. This confirmation is usually inferred from the changes sent by the server to the client. However, when the server is inactive for an extended amount of time (because none of the other clients made any changes), the number of local copies of changes that need to be stored by the client can grow unbounded. To prevent this, both the clients and the server need to periodically send explicit confirmation (i.e. empty change) messages to each other.

To achieve robustness in the presence of network failures, we could distribute the document over multiple servers. Clients can connect to any of the servers, and the servers keep each other synchronised using the same concurrency control algorithm as the clients. This requires the existence of a superserver, towards which the other servers act as clients.

To ensure performance under heavy load, we could allow clients to request the server to hand-off the document to another server. The server responds by sending its copy of the document to the other server, and informing the clients that they should send all future changes to the other server. Until the server has received confirmation from all its clients that they are now sending their changes to the other server, it forwards all the changes it receives to the other server. This prevents change messages from becoming lost during the transition.