Imperial College London

Software Engineering Group Project

Imperial College London

Department of Computing

Matterialize

Authors:
Matteo Bilardi, Andreas Casapu, Devesh Erda, Prabhjot Grewal, Eugene Lin, and Raihaan Rahman

Supervisor:
Dr. Nicolas Wu

January 12, 2021

Executive Summary

We find ourselves in an unexpected time where everyone from professionals to students to families are utilising video calling platforms to communicate with each other face-to-face. These platforms have surged in popularity and many provide the useful functionality of background removal and/or manipulation: a feature which can provide a personalised, or professional appearance and environment from home.

However, for those who use Linux, there is very little support for this feature, even on the most notable of video-calling solutions, without requiring investment into specific hardware or physical apparatus. This is the case even for developers wishing to create their own solutions: there are no obvious frameworks to use, the capture and re-streaming of a video feed is non-trivial to set up, and there are no evaluation tools for testing a solution. The only existing options are small side projects that run as a command line tool.

This is where our project, *Matterialize*, can make a difference. *Matterialize* is a unified development framework to create, test, and deploy background matting techniques on Linux, packaged with a modern and comprehensive graphical user interface. It has an open architecture for integration with third-party applications and we have already integrated three different examples of matting strategies to showcase the capabilities of the platform, one of which performs sufficiently well to merit real-world usage.

The backend has a clear and concise set of interfaces for developers, allowing them to integrate their background matting strategies into the application with ease. In addition, the framework's extensible design gives developers the option to create alternative frontend interfaces - such as a command line interface for a "headless" mode. The included GUI gives users the ability to dynamically change matter configurations, apply a wide selection of different background replacement effects as well as a suite of features valuable to developers during testing. The entire application is also packaged for easy installation on a range of Linux distributions.

1 Introduction

This report outlines the key design decisions that were made and implementation details for core functionality of the application and matting strategies that were explored. Also included is an evaluation of the achievements in both a quantitative and qualitative context, as well as options for further development.

1.1 Motivation

Image matting is an important and heavily used technique in image recognition, utilised in many practical applications, from object tracking in security videos, to deblurring and reconstructing hastily taken images, to restoring the otherwise lush colours that have been distorted in a hazy picture [1]. Regular individuals have also begun to use it with increasing frequency, often without even realising it. According to the Office for National Statistics, in April of 2020, 46.6% of people in the UK did some work from home, 86% of which as a result of the Coronavirus pandemic [2]. Many of these people have been using online communication platforms, such as *Zoom*, *Microsoft Teams*, and *Google Meet*, in order to coordinate and stay in touch, inadvertently being exposed to, or themselves using the virtual background features on offer. Such functionality allows one to replace their background with an image of their liking, or to blur and obscure it, while still remaining in frame themselves, and is, at its core, another application of image matting.

Not only are there many current methods to perform this, but it is also an active area of research [3, 4]. Furthermore, there exists a significant amount of interest from companies such as *NVIDIA* [5] and *Snap Inc* [6] in ever-improving solutions. This is because high quality, well performing, and generally applicable techniques are desirable for regular consumers and content creators alike. Despite the large-scale utility and applicability of this technology, there is little support for independent developers or computer vision enthusiasts that wish to get involved. Such individuals need to construct their own frameworks, handle the hardware interactions and evaluation methods themselves, alongside researching and implementing matting techniques.

Furthermore, there is a lack of consumer-oriented virtual background options on Linux distributions. For example, platform-agnostic solutions such as *Snap Camera* and *NVIDIA Broadcast* do not support Linux at all [7, 8], with the latter also requiring modern, proprietary hardware (*RTX 2060* or higher) - which can be inaccessible to many. Additionally, options for virtual backgrounds packaged within communication platforms are limited: *Microsoft Teams* does not provide this functionality on Linux and *Zoom* only offers chroma-keying [9]. The only solution we were able to find was an unintuitive, terminal-run application [10] implementing a virtual background approach, suggested by Benjamin Elder.

1.2 Core Objectives

From the developer's perspective, a crucial feature would be a simple method for a computer vision enthusiast to integrate matting strategies into the pipeline, with little knowledge regarding the rest of the application. This requirement can be summarised into a shallow learning curve for developers, while maintaining a high ceiling to possibilities for what they can achieve with the framework. In addition, users and developers should be able to interact with the application in real time via a graphical user interface, which should expose the core functionality of the image processing pipeline.

The application should be able to output its manipulated video feed into third-party applications without any discernible delay or loss in quality to the recipients. Additionally, running the application should not heavily stress the device in order to reduce the impact on the user's experience or ability to perform other tasks. Alongside the framework, there should be an implementation of a background matting technique that provides an output of adequate quality for users to use, as well as examples for developers to become accustomed to the infrastructure.

1.3 Achievements

Matterialize looks to ease the burden and make ventures into this area of computer vision more accessible to the developer by providing an installable, standalone, modular framework for developers to implement, integrate, and test their own matting strategies and techniques. The application allows for straightforward building and deployment, and includes an accessible, user-friendly GUI.

The UI can be detached, and substituted with ease, as the core of the software is able to communicate with other applications through a REST API.

The application focuses on Linux platforms, and provides a virtual camera feed which can be utilised within a user's communication software of choice, or with any application capable of capturing video devices. *Matterialize* comes prepackaged with an out-of-the-box matting algorithm of adequate quality, background alteration features, such as the ability to replace the background with images, videos, or blurring, and customisation options that allow users to optimise the matting algorithm parameters to their particular lighting conditions and environment.

2 Design and Architecture

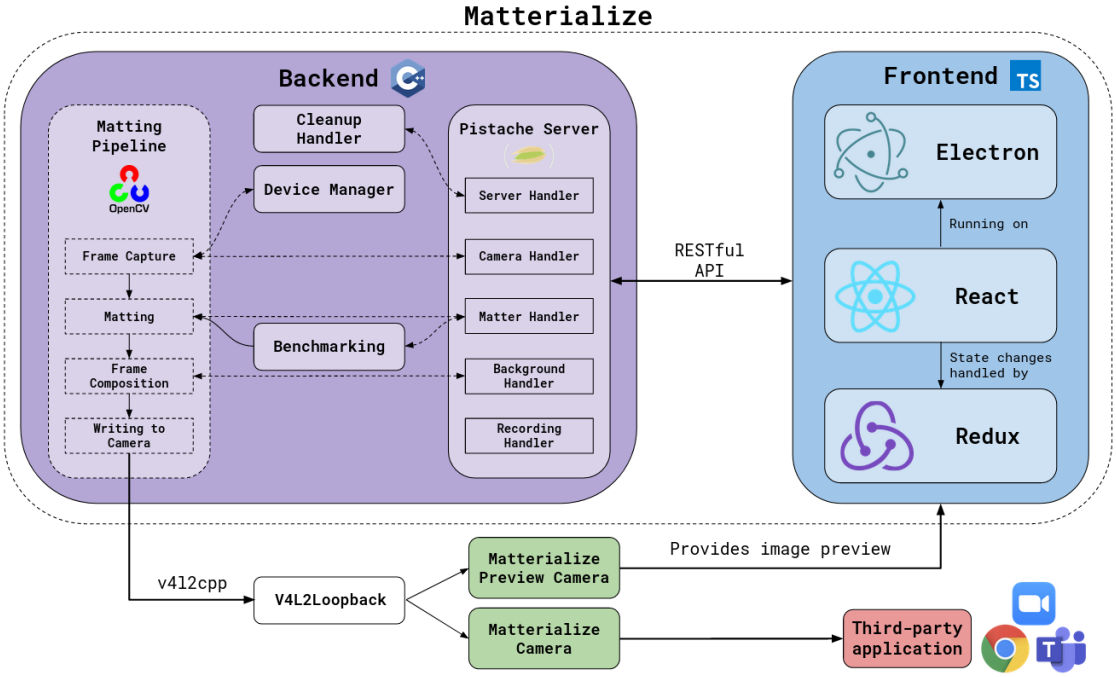


Figure 1: System architecture diagram displaying the main components of the application. The dotted arrows within the backend represent shared state between components.

2.1 Frontend-Backend Communication

In order to produce a unified application for use by developers and users, the platform required a frontend interface as well as a backend handling capture and manipulation of the video feed. Establishing a suitable communication method between frontend and backend was thus an essential step, with meaningful repercussions on implementation and user experience.

A RESTful approach was chosen, allowing for a simple and easily extensible interface for applications to interact with. This opens up the possibility for future integration with third-party applications including, for example, a "headless" mode, which would only run the backend, a command line interface that would parse and send the appropriate commands to the server, or a completely new frontend.

The alternative approach of running a GUI alongside the matting process in C++ using libraries such as *Qt* or *gtkmm* was also considered. It would have lead to a more monolithic design, with communication between the GUI and matting pipeline being restricted to shared state, protected by synchronisation primitives. This would have allowed more direct control via the GUI, but lacked the extensibility and flexibility of a REST interface. In addition, these libraries would limit the user interface to the components they provide, whereas a web application can freely use CSS and a variety of component libraries to make the GUI feel responsive, intuitive and modern.

2.2 Frontend

The team agreed to create the frontend using web based technologies, granting complete separation between the user interface and the backend logic. The frontend application enables the user to see a preview of the virtual camera, select their preferred matting mode, as well as view and interact with the customisation options and additional features available. When buttons are clicked, or options are selected, HTTP requests are made to the locally running REST server in the backend, which responds with the appropriate information and updates the state. Refer to appendix A for additional screenshots.

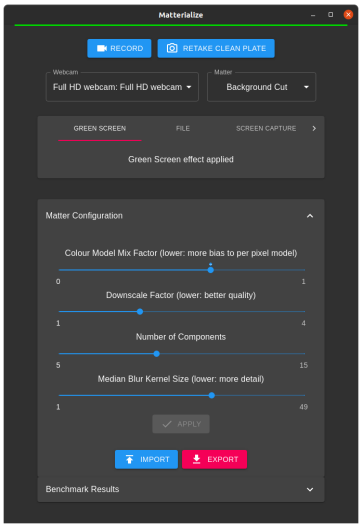


Figure 2: Matter configuration options

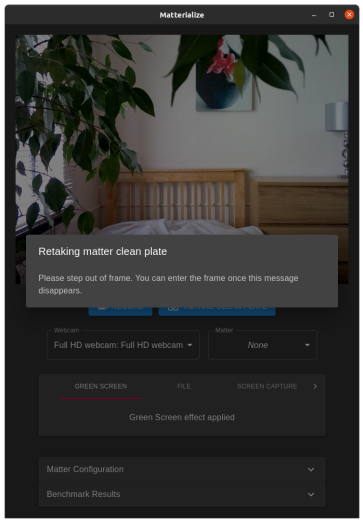


Figure 3: Clean plate retaking prompt

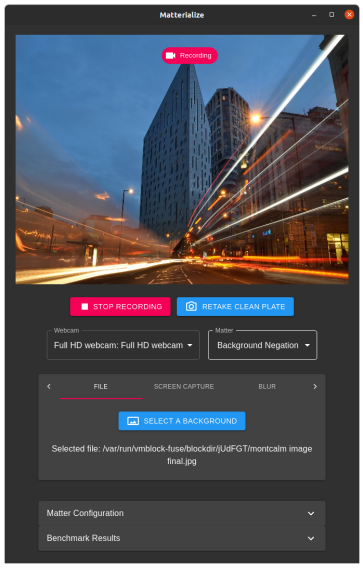


Figure 4: Video recording functionality

2.3 Backend

2.3.1 REST Server Handlers

Functionality of the server is split between different handlers, each handling requests for a given part of the matting pipeline. These are the following handlers and their respective responsibilities:

- Background Handler
  - Handles setting the background replacement/effects to be applied including static images, videos, screen capture and blurring.
- Camera Handler
  - Provides information on the current camera being used, available cameras, and handles setting which camera should be used.
- Matter Handler
  - Provides current state about what matter is being used, what matters are available, and the configuration parameters required for the matter.
  - Additionally, it handles requests to change matters, updates the configuration of a matter, and imports or exports configuration files.
  - The matter handler also handles requests to instruct the backend to take a new clean plate, and to benchmark the available matters.
- Recording Handler
  - Handles endpoints to start and stop recordings of the camera feed.

- Server Handler
  - Allows the frontend to control the running of the backend, so the backend can be shutdown via the frontend.

2.3.2 Matting Pipeline

To capture, manipulate, and restream a camera feed, the backend is structured as an image processing pipeline.

1. A frame is captured from the camera, then it is passed to a matter that may have been initialised with an image of a "clean plate" of the background. The matter implementation fulfils a simple `IMatter` interface that allows for developers to create their own matting techniques that can be easily integrated and used within the application.
2. When the frame has been processed by the matter, it returns a "mask" that is an image of the same dimensions as the input frame where pixels are given values from 0 to 255. White pixels, of value 255, represent parts of the frame identified as the foreground (thus should be kept) and black pixels, of value 0, represent the background (which should be manipulated).
3. Taking the original camera frame and the generated mask, the original image is altered according to the selected option. This includes blurring the background as well as background removal and replacement by static images, videos, or a live capture of the user's screen.
4. Once the final frame is generated, it is converted from BGR to YUYV [11] for better compatibility with video chat applications and then written to a virtual output camera for use by an application, as well as sent to a virtual preview camera feed that can be viewed in the frontend.

This structure of the matting process allows for developers to only need to be familiar with a small number of interfaces in order to get their matting techniques working and usable by the rest of the processing pipeline. In addition it is easily extensible, as one can add new functionality by inserting the feature somewhere along the pipeline. The pipeline itself does not track information about frames between the processing of consecutive frames reducing memory usage, therefore placing the responsibility of any temporal processing and state to the matters themselves - which can easily be integrated due to the design of how matters interact with the system.

3 Implementation

3.1 Frontend Stack

The frontend is written in TypeScript as it gives type safety within the project, allowing for clear and consistent interfaces which are less prone to human errors. React was used with *Material-UI*'s component library to rapidly prototype designs for a modern, user-friendly UI with high-quality, customisable and modular components.

Electron is an industry standard framework used to build the *React* app (a web application) into a native desktop application. It is used in applications such as *VSCode*, *Slack*, and *Discord* - applications all known to work across multiple Linux distributions [12]. This makes it possible to create a portable UI designed as a web application, without being concerned about compatibility across different Linux distributions.

In order to support new matters being implemented by an external developer, all configuration fields are generated on demand. Once the initial configuration values are loaded into the frontend, an internal map is generated to store metadata for these options. When the matter is changed by the user, the corresponding UI elements are generated with the parameters specified by the matter implementer. All of the aforementioned states are managed by *Redux*.

3.2 Backend Stack

The backend of the application is written in C++. The backend contains two main components: a web server to manage communication with a frontend UI, and a image processing pipeline that handles the various processes and components to create the desired output of the application (see section 2.3.2).

The primary reason for using C++ is the abundance of support in terms of computer vision libraries and resources. Many of them were written in C/C++, hence it was a natural choice in order to ease development and obtain appropriate tooling. Another consideration in choosing C++ was due to its performance advantage over most languages, which proved necessary once the matting strategies became more complex.

Python was also briefly considered due to its extensive use in the fields of machine learning and computer vision, and being very conducive to rapid iteration. However, due to existing concerns regarding potential performance overhead, it was decided against. Furthermore the team had more experience, and was therefore more comfortable, in C++.

3.2.1 REST Server

To facilitate communication between the user and the matting pipeline, a REST server using the *Pistache* framework for C++ is run locally alongside the matting process. *Pistache* was the framework of choice, as it was simple to integrate into the application. This was unlike more feature-rich alternatives which would have required a rewrite of networking and parsing code, such as *Asio* from the *Boost* libraries. The simplicity of *Pistache*'s interfaces and design allowed many members of the group to work with and build features that required communication between the frontend and backend, with minimal prior knowledge or research. Additionally, due to the server interfacing with the frontend locally with infrequent (generally small) requests, performance of the library was not a major concern. In fact, running the REST server has no significant performance overhead in comparison to the matting pipeline.

The REST server runs on a separate thread to the matting process, but only utilises a single thread to reduce unnecessary contention for resources and when updating state. When data is sent and received the server uses the JSON file format for ease of use, flexibility and compact data definitions. Refer to appendix B for a full listing of exposed endpoints and usages.

3.2.2 Computer Vision Library

The *OpenCV* computer vision library is used extensively throughout the matting pipeline. Grabbing frames, controlling the camera, internally representing frames as pixel matrices, and all linear algebra operations on matrices required for matting or compositing are performed via *OpenCV*. Due to its highly performant nature, this industry-standard library satisfied the real-time performance requirements of the product. This, in conjunction with the vast amount of documentation and resources available, made *OpenCV* an obvious choice for the project.

3.3 Camera Controls

In the backend frame processing pipeline, camera conditions are taken into account and normalised. In order to avoid unexpected driver interference, on startup, and upon taking a clean plate, the camera is allowed to auto-adjust to the optimal settings for the current conditions, before having its automatic exposure and white balance settings turned off. This is done through the options provided by the *OpenCV* library and results in a cleaner and more consistent matting result.

3.4 Managing Matters at Runtime

While the `MatterHandler` takes care of validating and parsing HTTP requests from the frontend client, the implementation logic for performing the requested updates to the matting strategy is delegated to the `MatterManager`. This object plays a crucial role in enforcing semantics that feel natural from the perspective of the user while keeping the internal state of the backend matters consistent with the frontend application.

Since initialising a matter might require training on initialisation data, such as a clean background plate taken by the user, we assume matter initialisation to be an expensive operation that should be performed only if absolutely necessary. Imagine, for example, that a user has initially requested in succession two different matter modes: Background Cut (which requires training) followed by Background Subtraction. If they then wish to go back to Background Cut, they should not have to wait for the training to be performed a second time. Similarly, they should not have to endure the long startup times that would follow from all matter modes being initialised before

they can use the application. Thus, the `MatterManager` maintains a `MatterState` for each matter selected and initialised so far to avoid spurious (re)initialisation.

On the other hand, a user might perform an operation that requires a matter to be reinitialised, e.g. taking a new clean plate if their background or lighting conditions have changed. In this situation, the most up-to-date background plate should be used by every matter mode that requires it as input. Still, since the minimum number of reinitialisations should be performed, the `MatterManager` reinitialises immediately only the currently selected matter, while all the other instantiated but not currently selected matters are marked for reinitialisation upon future selection by the user.

In summary, the `MatterManager` guarantees sensible and responsive semantics under updates that are enforced by lazily performing them so that they occur only if the user selects a matter mode for immediate use.

3.5 Matting Algorithms: Research and Implementation

During the initial iterations of the project, a sizeable amount of time and manpower was invested in researching the matting problem, obtaining an understanding of the solutions proposed by computer vision researchers, and attempting an implementation of a matting algorithm that would deliver satisfactory results. The importance of this process cannot be understated: by giving context for the problem domain, its complexities, and the practical implementation obstacles computer vision developers have to face, it provided motivation to build a truly useful tool.

Firstly, it was crucial to learn that matting (i.e. the task of classifying whether a pixel in an image belongs to the foreground or the background), is a mathematically under-constrained problem, so that any solution must take explicit steps to constrain it [13]. These might include making assumptions about image statistics, asking a human user for input, using depth information from an IR camera, etc. For example, if a matting algorithm assumes that the background is static and known in advance, this requirement is fulfilled by asking the user to take a clean background frame before starting an algorithm.

Then, to gain a better sense of the needs of matter developers for a variety of matting algorithms, several matter implementations were attempted.

Figure 5: Left-to-right, top-to-bottom: background subtraction, k-means, background subtraction, *OpenCV* face detection [14], background subtraction, Near-IR sensor

3.5.1 Background Subtraction

The initial algorithm choice was background subtraction, a simple matting algorithm that classifies a pixel at a given position in the current frame as belonging to the foreground if its difference with the pixel at that same position in a previously taken background plate is larger than some floating point threshold. Despite the simplicity of this algorithm, it can perform very well under good lighting conditions and a completely static camera. However, in the event of a sudden shift in camera positioning and lighting, or in the presence of similar colours between background and foreground, the quality quickly deteriorates.

3.5.2 K-Means

In order to improve the stability, the next attempt was to process the clean plate as well as each frame by performing K-means clustering over the colour space, before the background subtraction, in order to smooth out small deviations in colour due to lighting and shadow. Unfortunately, this approach proved ineffective, as the improvement in stability was overshadowed by the large drop in quality. Following this, was an attempt to add pixel location information to the clustering, which mildly improved quality, but required a large increase in the number of clusters, which caused significant performance issues. As such, the pursuit of this approach was ended and other avenues were considered.

3.5.3 Near-IR Sensor

Experiments were also done with the additional depth information produced by the infrared sensors on certain laptop models. In lieu of generating a mask based on the frame, the mask was generated based on the capture of a secondary camera, namely the NIR (Near-IR) sensor - this created a greyscale image that could be used as a mask, with basic thresholding, as well as synchronisation of frames due to the different frame rates between the two camera sources. This led to significantly more stable results, independent of lighting conditions and change of background. However, it had significantly worse quality along the edges, did not work entirely on foreground elements that were physically far away from the laptop, and required special hardware. This is due to how NIR cameras inherently work: by observing reflected light from an object, allowing for close facial details to be recognised, as well as not being affected by ambient lighting [15].

3.5.4 Background Cut Colour Model

While the experience gained from these algorithms was useful to further clarify the requirements of our product and to better understand its domain, it seemed important to implement a more sophisticated strategy. This was essential to increase awareness of the implementation experience from the perspective of computer vision researchers pushing the state-of-the-art, as well as to provide a satisfactory product to users of the final application.

When investigating the latest solutions proposed by researchers to the matting problem, a strong dependency of newer techniques on older ones quickly became apparent, with the theory and the implementation outline behind any current state-of-the-art solution easily spanning dozens of papers. In the light of this, the aim was to identify a matting strategy that was at once seminal enough to be representative of many of the challenges matter implementers have to address, fast enough to be run in real-time for video chat applications usage, and at a level of sophistication that would allow a team of developers with no previous experience in computer vision to reproduce it. Thus a partial implementation, with acceptable final results, of the basic colour model outlined in Background Cut [16], a paper with more than 338 citations on *Google Scholar*, was created.

The implemented strategy consists of a mixture of two colour models, one per-pixel and one global, to predict with which probability a given pixel belongs to the background. The per-pixel model uses a single multivariate Gaussian distribution for each pixel with parameters initialised using the vector with the BGR values of the corresponding pixel in the background frame as the mean of the distribution, and the diagonal matrix of variances of each colour channel measured on the 3×3 neighbourhood centred at that pixel in the background frame as the covariance matrix of the distribution. The global model uses a mixture of Gaussian distributions (GMMs) and is completely independent of the pixel's position as it is only based on the values of its colour components. The per-pixel model offers more accurate results (see figure 6) but if the assumption of static background is violated, e.g. if the lighting changes or the webcam is shifted, matting

quality becomes very poor (cf. figure 9). The global model, on the other hand, is more robust to changes in the background (see figure 11) but tends to lead to less accurate results e.g. if the colour of the foreground and background are similar. To provide a balance between the two models, their probabilities are mixed together according to a mixing factor (see figure 7 and figure 10).

Figure 6: Per Pixel Model Only (mixing factor set to 0%)

Figure 7: Mixed Model (mixing factor set to 50%)

Figure 8: Global Model Only (mixing factor set to 100%)

Figure 9: Per Pixel Model Only (slight vertical shift)

Figure 10: Mixed Model (slight vertical shift)

Figure 11: Global Model Only (slight vertical shift)

3.6 Matting Interfaces

For a computer vision developer, implementing a matter compatible with the application comes down to three steps:

1. declaring any configuration fields (`MatterConfigField`) they wish to expose for updates by end users
2. implementing the `IMatter` interface
3. implementing the `IMatterMode` interface

Below is an example highlighting the meaning of the relevant types and demonstrating how a computer vision developer can integrate a simple background subtraction matter within the application.

Background subtraction has a floating point threshold as a hyperparameter of its model. Thus, a developer might want to allow the user to modify the threshold field to better fit their particular background and lighting conditions. To do so, the developer must statically declare a `MatterConfigFieldDouble`, specifying an illustrative name to display to the user, a name for internal identification within a configuration file, whether updating the field should require the matter to be reinitialised, a default value for the field, min and max values, as well as a step size determining the granularity with which a user can change this field's value. Note that for a valid `MatterConfig` named `config`, the current value of this field can be accessed via `config.get(threshold_field)`.

```
const static MatterConfigFieldDouble threshold_field{
    "Threshold (lower: more background)",  // display_name
    "threshold",                           // internal_name
    false,                                 // reinit_on_update
```

```
    25.0,                                   // default_value
    0.0,                                    // min_value
    255.0,                                  // max_value
    0.1                                     // step_size
};
```

A concrete `IMatter` must maintain its own state, if required. Additionally, if initialisation data (such as a clean background plate) or configuration options are required, the constructor should take in a `MatterInitData` and `MatterConfig` respectively. To implement an `IMatter`, the developer should simply provide an implementation for the `background_mask` method, taking in the current video frame and producing a binary mask filtering out the background.

```
class BackgroundSubtractionMatter : public IMatter {
private:
  cv::Mat background;
  MatterConfig &config;
public:
  cv::Mat background_mask(const cv::Mat &video_frame) override {
    cv::Mat mask{video_frame.size(), video_frame.type()};
    cv::absdiff(video_frame, background, mask);
    cv::cvtColor(mask, mask, cv::COLOR_BGR2GRAY);
    double threshold{config.get(threshold_field)};
    double max_val{255.0};
    cv::threshold(mask, mask, threshold, max_val, cv::THRESH_BINARY);
    return mask;
  }

  BackgroundSubtractionMatter(MatterInitData &init_data, MatterConfig &config)
      : background{init_data.clean_plate}, config{config} {}
};
```

Finally, the developer must implement an `IMatterMode` by specifying a display name for the user to see when they pick this particular matting strategy, a method to initialise the concrete `IMatter` for this mode, whether a clean background plate is required for this matter, and a vector of pointers to all the matter configuration fields declared in step 1.

```
class BackgroundSubtractionMode : public IMatterMode {
  const string name() const override { return "Background Subtraction"; }
  IMatter *init_matter(MatterInitData &data, MatterConfig &config) const
      override {
    return new BackgroundSubtractionMatter(data, config);
  }
  vector<const IMatterConfigField *> config_fields() const override { return
      {&threshold_field}; }
  bool requires_clean_plate() const override { return true; }
};
```

To make the mode available within *Matterialize*, the developer should also instantiate it statically as a `constexpr` in `src/files/matting/modes.hpp` and add it to the `modes` array.

This concludes the implementation of the example matter. Upon recompilation of the backend, the new matting mode will be exposed to the frontend. Any user (including the developer) will be able to select the new mode, experiment with changing the threshold and immediately preview its effect, and benefit from all the additional features and effects provided by the application.

3.7 Effects

In order for the product to provide more value to a regular user, a decision was made to bundle a small suite of background effects. To support dynamic backgrounds, such as videos, background blurring, or desktop capture, the implementation was changed to have a mutable state which handles generation of images per frame. This is in contrast to an earlier implementation which only allowed for a single, static image.

For example, to support the use of videos, a persistent counter indicating the current frame of the source video would be stored. Once the background is switched to a video, the capture is reinitialised, and the frame counters reset. On every frame, once the mask is generated, the current frame is captured (using *OpenCV*'s video reader), and the counter is incremented. If the frame is empty, it indicates that the video has ended, in which case the counter is reset in order to loop the video.

Another factor in the shift to the new system was to simulate a depth-of-field effect, by blurring the user's background. This required the current frame from the webcam, which was previously inaccessible to the background handler, and thus also needed to be exposed to the background generator.

Finally, the use of desktop capture is performed with *X11* alongside *xrandr*, the latter being used to obtain data to support configurations with multiple screens. The former captures a single frame of the desktop [17] (with the dimensions specified by the results from *xrandr*).

The use of a persistent state allows configuration options to be stored and also allows for more complex backgrounds, permitting novel features and effects to be developed further.

3.8 Recording

Alongside the aforementioned features, the software solution also allows for built-in video recording. Upon pressing the appropriate button, or making a request to the endpoint, the application will begin recording the contents of the preview camera, which will be timestamped and saved to the `~/.matterialize/recordings/` directory. This functionality is provided through an *ffmpeg* child process, which is monitored, handled, and cleaned up by the backend. This feature has been added in order to give potential developers an easy avenue to store, transfer, and analyse the results of their implementations, even when outside the application.

3.9 Benchmarking

In order for matter implementers to gain valuable metrics on their solutions, a system of internal benchmarking utilities was developed. This utility automatically composes foreground images from *Adobe*'s Matting dataset [18], on top of background images from the *COCO* datasets [19]. By creating this system of automated benchmarks, developers are able to quantify their progress in terms of both performance (the processing time it required per image), as well as quality (by judging our predicted mask against the ground truth). While the system has its flaws (for example lighting being inconsistent between the subject and the room, as well as only working on static images), it allows developers to obtain an estimate of progress between development iterations.

This is also exposed through the frontend, allowing for developers to quickly compare different matting algorithms and implementations, as well as enabling users to decide which implementation would be optimal on their system.

3.10 Installation

Even though this software project is aimed at technically competent Linux users, having a sufficiently robust and reproducible way to build and install it is still crucial for its usability and usefulness. Thus, our application and its dependencies can be installed through a Makefile using the canonical command `make install` (tested on Ubuntu 18.04, 20.04, and Arch Linux). At the same time, it is important to keep the build system flexible enough for an expert user to easily extend the codebase or add further dependencies. To this end, our install script uses *vcpkg* [20], a command line C/C++ package manager that fetches and compiles the various libraries required. If a computer vision developer relied on a particular library to be installed for their matting strategy, they could easily specify it via *vcpkg* in the install script, bypassing the issue of possibly outdated or missing packages within a Linux distribution. Moreover, the Makefile allows the frontend and backend to be recompiled and installed independently of one another for faster build times during development or for swapping the provided frontend with a different one.

4 Project Management and Evolution

4.1 Agility

An agile methodology felt appropriate for organising the project due to the timeline of the predefined sprints. The team chose to implement scrum due to some members having experience working in a scrum team prior to this project. We utilised a *Jira* board as the backbone of our scrum structure, using it to manage tasks and issues during each sprint. The team met together every weekday for the daily scrum meeting, as a means to talk about our achievements in the previous day's work and the issues we faced in the process. This meant everyone would always be up to date with how everyone else is finding the work they have been assigned.

We held planning sessions before the start of every sprint, giving us a time to reflect on the previous sprint, to talk about the work we had achieved and any major issues we faced during the sprint. This was incredibly valuable as we used these insights to add tasks to our backlog for us to complete either in the upcoming sprint or one in the future. We then chose a particular goal for the sprint and selected tasks from the backlog accordingly. Members were allocated to each task dynamically according to priority, difficulty, and individual preference. When the size of a task was deemed significant, we found it beneficial to assign a pair of members to it or to reallocate other members to tackle it more effectively. Programming in pairs had the added positive effect of improving morale and team cooperation while working remotely and was thus the setup we favoured for the majority of important tasks. Furthermore, by having another perspective, we were able to avoid tunnel vision and instead discuss possible alternatives with each other.

4.2 Communication and Data

Live communication between team members for meetings, pair programming, and any kind of project related discussion that would have normally happened in person, occurred via calls on a *Microsoft Teams* group. *Teams*' calendar was used extensively to schedule biweekly meetings with our supervisor and daily scrums within the team, which automatically appeared on everyone's calendar and had thus excellent attendance. The use of breakout rooms during development was also significantly beneficial to our workflow and productivity. It allowed different pairs or sections of the group to work together, while allowing individual members to pop in and out of the rooms as necessary for quick questions, discussions, or assistance. Additionally, it averted the shortcomings of a single large group call where it can get confusing to address someone individually or work on different parts of the project at the same time. This emphasis on communication was a direct result of us choosing to implement scrum.

Moreover, it became clear early on in the development process that we were producing an ever increasing amount of information and collection of relevant resources that would have become difficult to record and query. Using *Slack* as our written communication platform of choice helped us to deal with that data by using different channels to collect meeting notes, research and discuss the implementation of particular features, dump code snippets, and build an archive of what we had been working on each day. *Slack* also allowed us to have a more direct channel of communication with our primary client.

4.3 Risks and Mitigations

During development of the framework, it became clear that performance (in terms of frame-processing times) was essential for the application to output usable results that didn't appear vastly out of sync with an audio source for example. Thus there became the risk that if the application itself had too much overhead and if the tech stack was not performant enough, it would not be usable for its purpose as a development framework. This risk was mitigated by the transition of the tech stack to using C++ on the backend and detaching the GUI to a separate application running using *Electron* and built using *React* for TypeScript.

In addition, after evaluating a variety of camera output resolutions, a decision was made to downscale any high resolution camera feeds to reduce processing times, as the quality differences were unnoticeable while using video calling platforms due to their own compression. These changes resulted in a very reasonable performance overhead for the application, which developers could implement their own strategies without being significantly handicapped by the performance of the

underlying framework, as can be verified by the near-real-time frame processing times of the simple background negation matting strategy.

Another risk was finding an appropriate pool of developers who would theoretically use our application, which we would need for feedback and testing. Due to the time and resource limitations of the development process, as well as the remote setting preventing in-person communication, it was incredibly difficult to find developers who could provide us with insights from their perspective of using the framework. To mitigate this, time was dedicated to implementing background matting strategies using the framework, providing insights into what the developers would be interacting with and their journey in using the application. These insights helped to influence some core design decisions and to create additional features such as dynamic changes to matter configuration options via the frontend. Getting feedback from an application user's perspective was viable and as such it was used to perform feedback-based iterations of the application.

4.4 Evolution and Challenges

4.4.1 Inexperience with Computer Vision

When implementing matting strategies and attempting to understand the perspective of potential developers, one hurdle that was faced was the new terminology and concepts that were unfamiliar to the group. For example, with the Background Cut paper that was used to implement the `BackgroundCutMatter`, a significant amount of probability calculations and concepts such as Gaussian Mixture Models required additional research and background reading to understand.

To overcome these challenges, members of the group spent time researching papers and finding useful resources online, and sharing them with other members of the group via *Slack*. This ensured knowledge and understanding was being shared amongst the group and that implementation details in the codebase could be properly understood.

4.4.2 Tech Stack Limitations and Changes

In order to provide developers with a platform that assists in the implementation of matting algorithms, and related computer vision techniques, development began with an initial phase of rapid prototyping. This was done in Kotlin, with an integrated frontend built using *TornadoFX*. During this phase, time was taken to become familiar with some common computer vision libraries and terminology, as well as how to manipulate and emulate video capture devices.

Following the prototype, time was invested into restructuring the codebase into a more modular and flexible design. Simultaneously research also began into effective matting techniques. By understanding matting techniques and attempting to implement more advanced strategies, a better understanding of a developer's perspective and what could be done to aid them was obtained. This included discovering features and functionality that simplified the development process, what metrics could be used to evaluate matting techniques, and external influences that could impact the quality of background matting such as camera exposure and lighting. These insights were then used to influence the design and architecture of the application, as well as contributing to the addition of various features such as fixing camera exposure levels.

With these insights, it became clear that with the current design it would be difficult to develop the prototype into the final application. In addition, the current tech stack was becoming a limitation for future development as most resources and libraries required for the software solution were only available in C++ and Python, and the matting implementations also had sub-par performance in Kotlin. There was a heavy reliance on C/C++ libraries which required the use of the Java Native Interface (JNI) to bridge the gap between languages. Additionally, due to JNI limitations with C++, an unsustainable amount of development time was being spent writing wrappers of C/C++.

Considering these aspects, the decision was taken to re-architect the design of the system and port the useful functionality to a more appropriate tech stack. The core of the backend was ported to C++, and a new frontend was created using TypeScript. This removed the burden of using incomplete, or custom library wrappers, and the application gained a noticeable performance boost, which proved essential to maintain adequate frame-processing times once the background cut algorithm was extended. Throughout the rest of the development process, the decision yielded additional benefits and in general allowed for a more frictionless development process, a cleaner design, and a better user experience within the GUI and for matter development.

After the completion of porting, code cleanup, and integrating the improved frontend, the focus of development shifted to providing additional functionality and features that helped the overall user experience of setting up and using the application. This included features such as an installation script, and benchmarking capabilities (accessible via the frontend), as well as expanding the suite of virtual background features. With external users to test the application, it was possible to make further feedback-driven changes to the application. These included switching the default theme from light to dark, and creating avenues for developers to be more descriptive with the matter configuration options displayed to the user.

5 Evaluation

We believe we have achieved a satisfactory end product, in terms of user needs, which has the potential for deployment within a reasonable time-frame. Throughout development, we have used our product during our milestone meetings with our supervisor, as well as during our regular meetings, in order to monitor its state, identify issues that regular users might encounter, and test it on different environments and conditions.

5.1 Quantitative Evaluation

Due to the nature of the project, unit testing was limited, as many of the core elements are either UI-related, or require the live processing of frames. These aspects were difficult to reliably test in an automated way, however, as we frequently made use of the application and kept track of any issues that arose, we believe we were able to substitute in part the reliability usually provided by unit tests with our own, manual testing.

In addition to our manual testing, the benchmarking suite we developed (detailed in section 3.9) allowed us to track performance between iterations, as well as different hyperparameters, of an algorithm.

5.1.1 Static images

The benchmarking suite was used to test background negation against background cut, the results are as expected: background cut (on the default configuration) obtains a higher accuracy overall (95.07% compared to background negation's 92.53%), at a noticeable performance penalty (around 30 to 60 milliseconds per image). However, it is important to note that accuracy alone is an insufficient metric for matting quality: background negation generally obtains acceptable accuracy scores, but it can produce significant artifacts depending on the scene, whereas the masks generated by background cut are smoother (cf. figure 13 and figure 14).

Figure 12: Original Mask (Ground Truth)

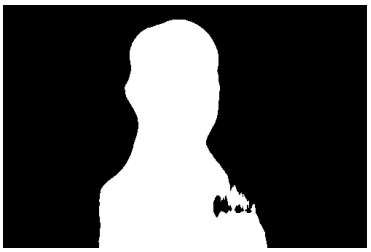


Figure 13: Generated Mask (Background Cut)

Figure 14: Generated Mask (Background Negation)

5.1.2 Video feed

In order to test the performance on a video, an 8 second, 30FPS clip was recorded in front of a greenscreen with room lighting set to 5500 Kelvin, to emulate daylight. A ground-truth background mask was obtained by chroma-keying in *Adobe After Effects* (with the use of *Keylight* [21]). The foreground of the video was then composed on top of a static background to generate a testing input video feed. Applying the background cut algorithm over the generated footage, an accuracy of above 98.60% was achieved in every frame when comparing the generated mask against the

ground truth. While it is necessary to acknowledge that this result is not representative of real footage, it can still be used to compare the effectiveness of different algorithms.

5.1.3 Performance Improvements

In contrast to the initial Kotlin prototype, there is a significant improvement in frame processing times for the standard Background Negation matter. Note that this matter was chosen as the implementation remained the same between the two versions, and therefore provided a fair comparison. The time between each consecutive frame in the Kotlin implementation, on average, was 82.47 milliseconds, with a standard deviation of 7.20 milliseconds - with some frames taking as much as 114.00 milliseconds. On the other hand, the same statistics in C++ were 32.70 milliseconds and 2.96 milliseconds respectively. These results not only show that the performance is significantly better in C++, but also more consistent which reduces the noticeable 'stutters' present in the Kotlin prototype.

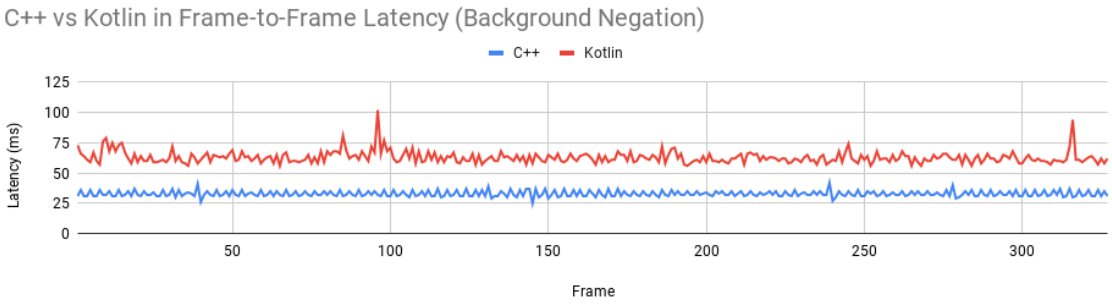


Figure 15: Frame-to-frame latency of background negation implementations in both C++ and Kotlin; lower times are desirable, but consistent times are also crucial

5.2 Qualitative Evaluation

We met with our supervisor, who also served as our primary user, every two weeks to discuss the progress of the project, as well as to determine the tasks and direction for the following sprint. At every such meeting, our supervisor was satisfied with our progress and addition of features, indicating that they met his standards and requirements. In addition to this, we have also decided to showcase our product to external users, in order to get a more diverse perspective and supplementary feedback on our work.

The feedback received, both external and internal, for the initial Kotlin implementation provided a good idea of the direction we needed to take. One of the main issues we noticed was the performance impact, when the frame rate was much lower than the camera's, thus leading to a noticeable delay in our generated video feed - informing us that major optimisations would be required. From the perspective of a regular user, there was little guidance on how the application should be used; with one user remarking "I think it would be useful to actually put some instructions, because I don't actually understand how this works". Furthermore, users also remarked on the lacking design; suggesting that we don't display both cameras, with another user providing suggestions on how the options should be presented.

Regarding our latest version, the users remarked that it is "more aesthetically pleasing", when compared to the earlier Kotlin version they had been exposed to. Another user said that they appreciate the increased instructions and notifications within the application when using certain features, exclaiming that "It tells you what to do this time!". Overall, our users appreciated the progress we have made on the UI, and found it to be more appealing, clearer and easier to use. However, our testers also identified a couple of pain points that they would like to see improved. In particular, they found the matter customisation options somewhat unclear in their functionality, being cited as saying that "for normal users it doesn't mean much". They also struggled at times to discern the use of certain layout elements, and were completely unaware of certain features, which were not appropriately signposted. This suggests the need for a tutorial, to guide the user

through our set of features, as well as tooltips, that could provide more context about how to use the functionality present.

Finally, our users commented on potential additions that they would like to see. While they were happy with the breadth of elements designed to support background replacement, they suggested that users might require better quality foreground detection, in order to fully make use of the application. One user also brought up the idea that a dark theme would be very useful, as the current light theme shines too brightly, affecting camera exposure when a clean plate is taken. From the perspective of a potential computer vision developer, a user noted that the "benchmarks that are currently present are somewhat simplistic", mentioning how more detailed information and better customisation options for evaluation would be necessary.

Due to the niche audience for one of our objectives, namely computer vision enthusiasts, we were unable to obtain extensive feedback from their perspective. However, with more time and resources, our testing methodology would follow the lines of asking our testers to implement their own matters using our framework (whilst observing their uses of our exposed interfaces) and observing their use of the available user interface to tune settings. Another metric that would be useful to measure is the amount of code that would need to be modified in the existing codebase, should the information provided by the current matting interfaces be insufficient. If the modifications were found to be common among testers, it might be worth exposing them in a generalised manner through our interface. This methodology was one we closely followed ourselves; when developing our bundled matters, we realised the importance of exposing certain configuration parameters to allow a user to tweak settings for their conditions.

Following the feedback, we have implemented the dark mode and made it the default theme of our application. We have also extended the matter configuration functionality to allow developers to provide a more verbose descriptor so as to inform the user of the impact of changing an option.

Overall, both our primary user, as well as our external testers were satisfied with the progress we have made on the application, as well as with the state of the final product. In the case of the external testers, which had more time to utilise the functions themselves, they provided useful insight about potential areas of improvement and expansion that would make for a smoother experience and add value for our target audience.

6 Ethical and Legal Considerations

Due to the nature of our project, there are not many ethical issues to take into consideration. For regular users, we provide a benign method to virtually replace their background, making no guarantees about what may be obscured, and we are open about the fact that we cannot promise that any sensitive items or information in the background will be hidden. For example, no guarantees are made that the matting techniques will be able to obscure the identities of people in the background which may be a requirement in cases of news reporting or with filming children. Users are not expected to be able to use this functionality in any malicious way that would not be possible in other applications that offer similar functionality. Furthermore, no guarantees can be made about the legality of the substituted background, therefore any issues related to copyrighted content would be the responsibility of the user.

From the perspective of developers, *Matterialize* is offered as a platform to implement, test and integrate their own background replacement techniques. A malicious individual could potentially write dangerous code and present it as a matter within the application, however it is infeasible to police such activity, and users and developers are encouraged to approach any piece of software, not only *Matterialize*, with caution. The application should not open any user to potential vulnerabilities, nor does it track, store or transfer any kind of sensitive data, as it only runs locally, requires no connection to any network, and does not make use of elevated privileges, beyond installation.

One legal concern to take into account relates to the use of a C++11 friendly mirror of GraphCuts [22, 23, 24, 25]. While the library is not currently in use, part of the full background cut algorithm requires performing energy minimisation on a particular equation. A fast and effective method to achieve this is implemented in the aforementioned library and it is the only one of its kind that could be located. However, the library license only permits use for research purposes, which means that a separate license agreement would need to be negotiated with the creator, or a suitable alternative would need to be found should the application be used in a commercial setting.

Another legal consideration is concerned with the use of the *Adobe* Deep Image Matting dataset. Since this set is included in the current benchmarking suite, the application cannot be used for

commercial purposes, nor can the images be redistributed (and therefore substitutions will need to be made for the benchmarking functionality). These restrictions also apply to the generated compositions, as the new images fall under the category of derivative images. Additionally, should other methods involving training models based on this set be attempted, these models would only be able to be distributed under the condition of non-commercial use.

7 Conclusion

Matterialize fulfils the core objectives that were outlined for the problem that it intended to solve (see section 1.2).

The matting interfaces provide a simple way for developers to integrate their own matting techniques, while tunable parameters, which are modifiable in real-time through the frontend, allow for more advanced customisation and experimentation for users and developers alike.

The frontend exposes an array of different features provided by the backend pipeline, including background replacement, blurring, desktop capture, recording, and benchmarking. Looking at our qualitative evaluation, users were pleased with the appearance and suite of features available within the GUI, with some 'quality of life' improvements suggested as future work.

For real-world usage, our application is able to output to a virtual camera feed readable by third-party applications such as *Microsoft Teams* and *Google Chrome*. Additionally, when assessing the performance impact of the application, we observed that the main computational load was a result of the matting technique being used rather than the application or the pipeline, which is highly desirable.

While the quality and performance of the bundled matting techniques are of an acceptable standard, further improvements can still be made. Matting quality could be improved by continuing the implementation of the Background Cut paper [16] to include the energy minimisation steps, thus completing the basic model. It would also be beneficial to explore alternative solutions to background matting, including concepts seen in more recent research papers [3], such as deep neural networks.

Future development could also involve the implementation of an alternative frontend, such as a command line interface, or direct integration into third-party applications. There is also room for further improvements to the user experience as a whole, including a user tutorial and tooltips, better camera control settings, more customisable configuration files, and presets.

References

[1] Dylan Swiggett. Image matting and applications.

[2] Alastair Cameron. Coronavirus and homeworking in the uk: April 2020, Jul 2020. URL https://www.ons.gov.uk/employmentandlabourmarket/peopleinwork/employmentandemployeetypes/bulletins/coronavirusandhomeworkingintheuk/april2020.

[3] Soumyadip Sengupta, Vivek Jayaram, Brian Curless, Steve Seitz, and Ira Kemelmacher-Shlizerman. Background matting: The world is your green screen. *arXiv preprint arXiv:2004.00626*, 2020.

[4] Minglun Gong, Liang Wang, Ruigang Yang, and Yee-Hong Yang. Real-time video matting using multichannel poisson equations. In *Proceedings of Graphics Interface 2010*, pages 89–96. 2010.

[5] Gerardo Delgado. Introducing the nvidia broadcast app, Sep 2020. URL https://www.nvidia.com/en-gb/geforce/news/nvidia-broadcast-app/.

[6] Snapchat. News – introducing snap camera, Oct 2018. URL https://snap.com/en-GB/news/post/introducing-snap-camera.

[7] Snapchat, Jan 2021. URL https://support.snapchat.com/en-US/article/snap-camera-faq.

[8] NVIDIA. Download the nvidia broadcast app. URL https://www.nvidia.com/en-gb/geforce/broadcasting/broadcast-app/.

[9] Benjamin Elder. Open source virtual background, Apr 2020. URL https://elder.dev/posts/open-source-virtual-background/.

[10] Pangyu Teng. pangyuteng/virtual-background, Nov 2020. URL https://github.com/pangyuteng/virtual-background.

[11] Laurent Itti. src/jevois/image/rawimageops.c source file, 2016. URL http://jevois.org/doc/RawImageOps_8C_source.html#l01038.

[12] Galabina N. 10 most popular electron apps of 2019, Dec 2020. URL https://wiredelta.com/10-most-popular-electron-apps-2019/.

[13] Jue Wang and Michael F Cohen. *Image and video matting: a survey*. Now Publishers Inc, 2008.

[14] Shekhar Pandey. Face detection using haar cascades, Aug 2020. URL https://www.studytonight.com/post/face-detection-using-haar-cascades.

[15] Ted Hudek and Joshua Baxter. Windows hello face authentication, Feb 2017. URL https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/windows-hello-face-authentication.

[16] Jian Sun, Weiwei Zhang, Xiaoou Tang, and Heung-Yeung Shum. Background cut. In *European Conference on Computer Vision*, pages 628–641. Springer, 2006.

[17] Brandon. c++ fast screenshots in linux for use with opencv, Jun 2016. URL https://stackoverflow.com/a/24988220.

[18] Ning Xu, Brian Price, Scott Cohen, and Thomas Huang. Deep image matting, 2017. URL https://sites.google.com/view/deepimagematting.

[19] T Lin, Michael Maire, Serge J Belongie, Lubomir D Bourdev, Ross B Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. arxiv 2014. *arXiv preprint arXiv:1405.0312*, 2014.

[20] Microsoft. microsoft/vcpkg, Jan 2021. URL https://github.com/microsoft/vcpkg.

[21] Charles Yeager. How to key green screen footage in after effects, Jan 2020. URL https://www.premiumbeat.com/blog/after-effects-green-screen/.

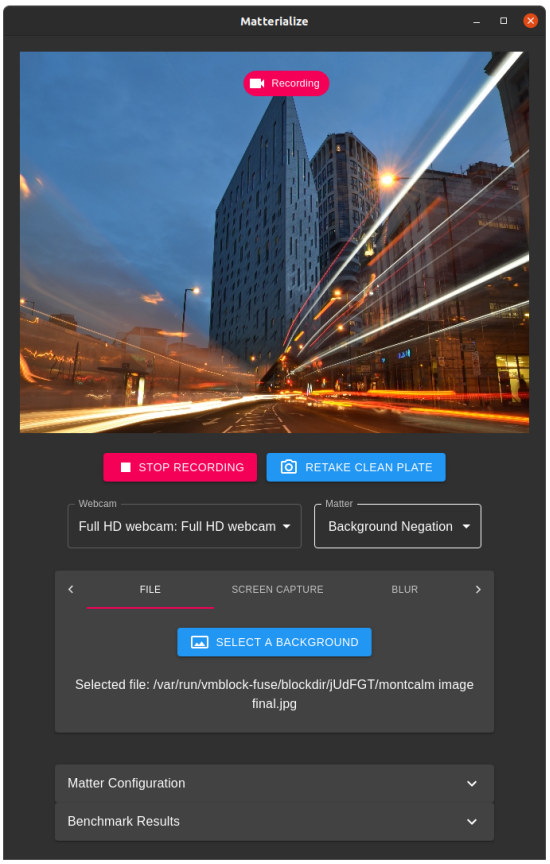
[22] Marco Livesu. mlivesu/graphcuts, Nov 2018. URL https://github.com/mlivesu/GraphCuts.

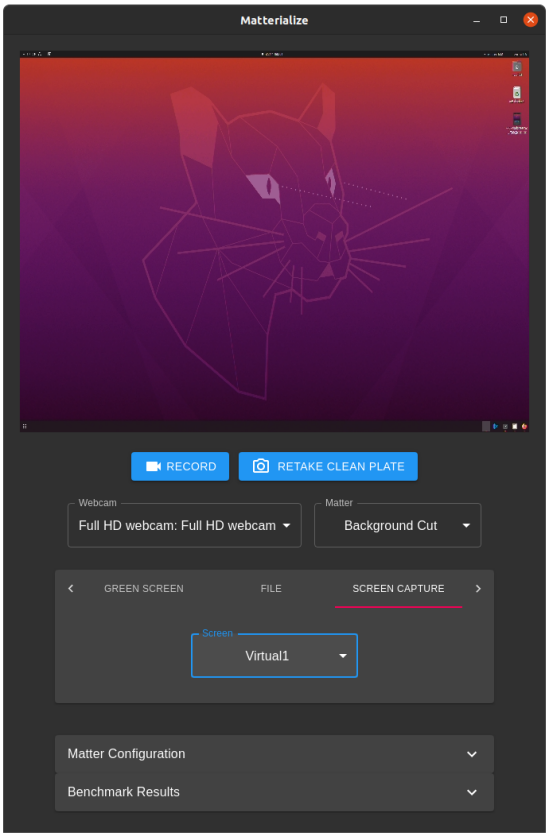
[23] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on pattern analysis and machine intelligence*, 23(11):1222–1239, 2001.

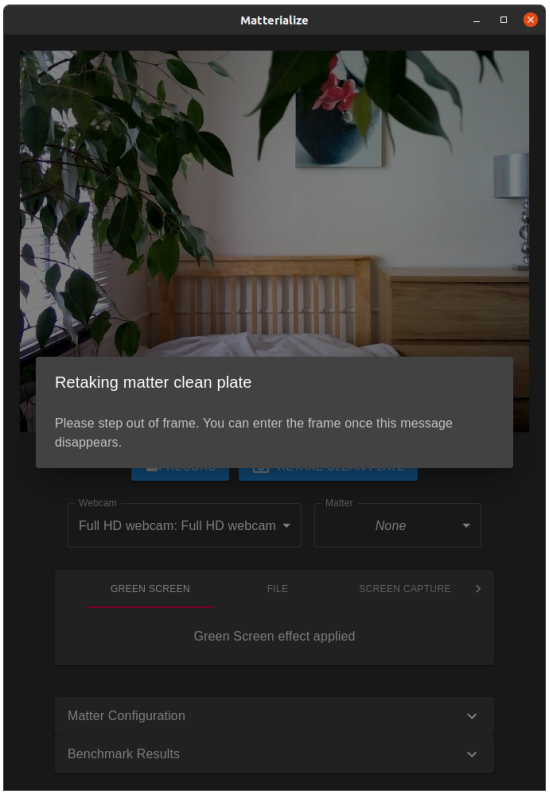
[24] Vladimir Kolmogorov and Ramin Zabin. What energy functions can be minimized via graph cuts? *IEEE transactions on pattern analysis and machine intelligence*, 26(2):147–159, 2004.

[25] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

A UI Screenshots







B REST API Endpoints

`/background/blur` `POST`

Used to switch the background effect to blurring, as well as to update blurring radius.

query parameter	type	description
`size`	integer	box blurring radius, must be an odd number

`/background/clear` `POST`

Used to switch the background effect to a solid green screen (hex value `#00FF00`).

`/background/desktop` `POST`

Used to switch the background effect to a desktop capture.

query parameter	type	description
`desktop`	string	display name (identifier), can be obtained from xrandr

`/background/desktop/options` `GET`

Used to obtain a list of desktops and their respective details.

response parameter	type	description
`devices`	array	array of objects containing display information
`name`	string	identifier of a display from xrandr
`x`	integer	x coordinate of top left corner
`y`	integer	y coordinate of top left corner
`width`	integer	number of horizontal pixels
`height`	integer	number of vertical pixels

Example response;

```
{
  "devices":[
    {"name":"eDP-1","x":3840,"y":0,"width":1920,"height":1200},
    {"name":"DP-1","x":0,"y":0,"width":1920,"height":1080},
    {"name":"DP-2","x":1920,"y":0,"width":1920,"height":1080}
  ]
}
```

`/background/set` `POST`

Used to set the background to a static image file.

query parameter	type	description
`file_path`	string	path to an image file

`/background/video` POST

Used to set the background to a static video file.

query parameter	type	description
`file_path`	string	path to an video file

`/camera/current` GET

Used to identify the current camera in use by the server.

response parameter	type	description
`dev_num`	integer	identifier of a video device from v4l2ctl
`available`	boolean	determines whether the camera is use by another application

`/camera/options` GET

Used to obtain a list of cameras and their respective details.

response parameter	type	description
`devices`	array	array of objects containing camera information
`dev_num`	integer	identifier of a video device from v4l2ctl
`name`	string	human-readable name for a given device

`/camera/set` POST

Used to update the capture source for the processing pipeline.

query parameter	type	description
`dev_num`	string	identifier of a video device from v4l2ctl

`/cleanplate/take` POST

Used to instruct matting pipeline to reinitialise with a new clean plate (required as a precondition for certain matting algorithms).

`/matter/config/update` POST

Used to update configuration parameters.

query parameter	type	description
`matter`	string	name of matting algorithm
`config`	object	updated configuration parameters, with fields obtained from `/matters/config`

`/matter/current` GET

Used to obtain the matting algorithm currently in use.

response parameter	type	description
`matter`	string	name of matting algorithm

`/matter/options` GET

Used to obtain a list of available matting algorithms.

response parameter	type	description
`matters`	array	array of objects containing matter information
`name`	string	name of matting algorithm

`/matter/set` POST

Used to change the matting algorithm. Note that this will fail with error 412 (precondition failed) if a clean plate does not exist, but the algorithm requires one.

query parameter	type	description
`matter`	string	name of matting algorithm

`/matters/benchmark` GET

Used to export raw statistics from the benchmarking utility.

response parameter	type	description
`accuracy`	double	accuracy of matting algorithm on a set of samples
`run_time`	integer	processing time of algorithm in milliseconds

Example response;


```
{
  "Background Cut":{"accuracy":0.9725944010416666,"run_time":68},
  "Background Negation":{"accuracy":0.9635384114583333,"run_time":0},
  "None":{"accuracy":0.32181966145833335,"run_time":0},
  "OpenCV":{"accuracy":0.32181966145833335,"run_time":14}
}
```


`/matters/config` GET

Used to obtain current configuration values and parameter metadata of matters. This functionality allows for simple configuration and tuning from a UI. Note that in the example below, `threshold` is the name of a configuration field and matches with `name`.

response parameter	type	description
`field_info`	object	contains metadata on a given parameter
`default_value`	double	base configuration value, specified by the developer
`display_name`	string	text displayed on the user interface
`max`	double	upper bound for parameter
`max`	double	lower bound for parameter

must_reinit_on_update	boolean	states whether a reinitialisation is required
name	string	unique identifier for a configuration field
step_size	double	determines granularity of changes
value	double	current value of parameter

Example response;


```
{
  "Background Cut": { ... },
  "Background Negation": {
    "threshold": {
      "field_info": {
        "default_value": 25.0,
        "display_name": "Threshold (lower: more background)",
        "max": 255.0,
        "min": 0.0,
        "must_reinit_on_update": false,
        "name": "threshold",
        "step_size": 0.1
      },
      "value": "25.000000"
    }
  },
  "None": null,
  "OpenCV": null
}
```


`/matters/config/export_file` POST

Used to export the current configuration into the default directory.

`/matters/config/import_file` POST

Used to import a configuration file from default directory to use as the current configuration.

`/recording/start` POST

Used to initialise the child `ffmpeg` process to begin recording.

`/recording/stop` POST

Used to terminate the child `ffmpeg` process to end recording.

response parameter	**type**	**description**
path	string	directory in which the video has been recorded to

`/shutdown` POST

Used to gracefully terminate the process.