ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Algorithms
and Didactics

Implementing GPanel: A Coordinate Graphics Library for WebTigerPython

Mark Csurgay

Bachelor's Thesis

2024

Supervisors: Prof. Dr. Dennis Komm
Alexandra Maximova
Clemens Bachmann

Abstract

WebTigerPython is an innovative online Python programming environment that introduces and teaches programming to students of various ages. This thesis dives into multiple approaches to enable JavaScript execution in Python code, facilitating the use of PixiJS, a widely adopted JavaScript library, for rendering graphics and shapes to the browser. The overarching objective is to identify the most suitable approach to implement GPanel, a coordinate graphics library, in WebTigerPython.

The merits, limitations, and performance metrics are weighted for each approach to ascertain the most promising strategy for the final implementation.

The final approach allows for executing JavaScript functions efficiently and cleanly, making this new method more performant than the approach used in the already existing GTurtle enactment. It executes any code written in a separate thread, allowing for the website interface to continue functioning while the code runs in the background.

Acknowledgment

I would like to express my sincere gratitude to several individuals and organizations who have supported me throughout the process of completing this bachelor thesis.

First and foremost, I am deeply thankful my supervisors Alexandra Maximova and Clemens Bachmann for their invaluable feedback on the thesis and their quick help in any coding related questions. Their constructive criticism and insightful suggestions have significantly contributed to the quality of this work.

I extend my heartfelt thanks to Dennis Komm for allowing me to conduct my research and complete my thesis in his research group. Their support and resources have been indispensable in the successful completion of this endeavor.

Contents

Chapter 1

Introduction

1.1 Background

TigerJython is a specialized programming environment written with the aim of simplifying the learning process of Python. It provides a user-friendly interface that consists of two main parts: a code editor with a 'start' and 'stop' button and a canvas that can be used for visual programming. The editor allows students to write Python code and execute it all in a standalone application. Any errors caused by users are clearly marked inside the code and shown on the correct line of the source code. The canvas allows students to draw pictures using code. The two main ways of drawing include using GTurtle, consisting of a virtual Turtle that can be moved around to draw wherever it goes, and GPanel, a graphics library, introducing a more mathematical approach to drawing with a coordinates system.

TigerJython was part of Tobias Kohn's dissertation in 2017, it started as a solid initial version, showcasing the possibilities in a programming environment designed for students. It uses Jython, which allows for Python code to be executed in Java and Java to be executed in Python. From a practical point of view, Jython allows for TigerJython to be shipped as a single executable Java application, enabling it to be used on any system [20].

A problem with TigerJython is that it has to be explicitly downloaded to be run, making it a big hurdle for students to use outside of a school lesson or on mobile devices. Nicole Roth (formerly Trachsler) developed WebTigerJython for her master's thesis [38]. This browser-based implementation of TigerJython allowed the programming environment to be used on any device, including mobile devices.

The latest version, WebTigerPython (formerly WebTigerJython-3), was initially implemented by Clemens Bachmann for his master's thesis [2]. It aims to fix the limitation imposed by WebTigerJython of not being able to use a vast amount of Python libraries without first having to port the library over for use in the browser.

The new base used for letting Python code run in the browser also means the main libraries used for drawing on the canvas, mainly GTurtle and GPanel, had to be rewritten, for the most part, to allow for them to work in the new WebTigerPython application. GTurtle was implemented by Julia Bogdan for her bachelor's thesis [3]. The GPanel library had not yet been ported over to WebTigerPython, adding a motivation for this thesis.

1.2 Main Contribution

This thesis foremost explores multiple approaches for working together with Pyodide and PixiJS to draw on an HTML canvas in an interactive fashion. The methods are tested for functionality, performance, and developer experience. A central goal was to get a strong base running with which future WebTigerPython graphics-related features can be built. To this end, a primary goal was to bring a large part of the graphics logic into the Python side of the code base, facilitating future development.

As a second step, this thesis implements the GPanel graphics library for full use in WebTigerPython. The functionality comprises a graphical editor that can draw various shapes, listen to events, and create animations. The complete function usage documentation can be found in the WebTigerPython documentation.

This thesis details the design choices behind the final implementation in Chapter 4. Finally, the many tested approaches, the ideas behind them, and, in detail, why each approach did or did not work are documented.

Chapter 2

Related Work

To understand how similar projects dealt with issues we faced, I looked into other projects, reading their documentation and inspecting their source code to see how certain features were made to work.

2.1 GPanel

GPanel was initially implemented in the desktop version of TigerJython. It is a graphics drawing library that supports different functions, from drawing simple shapes like rectangles or circles to listening to specific keyboard events and creating games.

This thesis builds a GPanel implementation for WebTigerPython, intending to keep all the functionality identical to the original.

2.2 GTurtle

A similar work to GPanel is the more well-known GTurtle. GTurtle builds on Turtle Graphics, which was first used in an educational context together with the Logo programming language [25]. This initial version used the Logo programming language to instruct a Turtle to move around and draw. Even though today's GPanel uses Python as the base programming language, most of the functionality stays the same. Moving the Turtle and drawing still happens with very similar commands in modern adaptations.

The older versions used in TigerJython or WebTigerJython were less relevant, so my main focus was on the initial version in WebTigerPython as implemented by Julia Bogdan [3]. This implementation looks at an initial approach for running Python in the browser to draw on the HTML canvas.

The core functionality behind the GTurtle implementation relies on a "Dispatch Action" function, which will be detailed in Section 5.2 and referred to as `dispatchAction` in this thesis. At its core, the `dispatchAction` wrapper function takes a command as a string and some arguments, which are sent from the web worker to the main thread as a message. The web worker here allows for Python code to run in a separate thread and enables the main thread to handle the website interface, preventing it from freezing up. Pyodide handles a big part of passing

messages between the two threads, making the whole implementation more organized and error-resistant.

This approach had multiple downsides as it was tedious to maintain. It required developers to implement the logic for every function in two parts. The Python side was executed in the web worker, and the functions needed to be callable in WebTigerPython. The JavaScript side was in the main thread, containing the Turtle functionality's relevant logic. A second downside was the performance overhead for sending messages from the web worker to the main thread and the response back to the web worker. Still, the GTurtle implementation served as a good starting point for approaches to look into.

2.3 Pyxel

An extensive and well-established library that also depends on Pyodide is called Pyxel [19]. Pyxel is a game engine for writing games in Python. Apart from its retro theme and limited color palette, its main attraction point is the ability to run it on any system and in the web browser. The latter functionality allows games to be hosted on a web server and enables mobile or desktop users to play the games without handling any Python install.

Pyxel uses a different tech stack than is used in WebTigerPython. The higher-level functionality and interface of their code are built using Python. The central part of the code and the core engine are made with Rust, resulting in a high-performance execution. The Rust code is compiled into web assembly, allowing for it to be loaded and used in the browser using JavaScript.

Since Pyxel is a Python library, it needs to have a way for Python code that was written to be executed during runtime. For that, the authors depend on the Pyodide Python library, allowing Python code to be executed in the browser. Section 3.2 will go into more detail about Pyodide.

The main interest for Pyxel was that it also used Pyodide. In their case, Pyodide ran in the main thread, which was different from how, for this thesis, the requirement was to execute Pyodide and Python within a web worker. Using their own Rust engine based on WebAssembly also meant they used a different graphics library implemented in Rust.

2.4 Enpyre

Enpyre [10] is another game engine for Python. It is entirely made for use in the browser, allowing a game created with this engine to be executed quickly on any operating system. The tech stack of Enpyre includes ReactJS, PixiJS, and Pyodide. Enpyre's website is built using ReactJS, a JavaScript UI framework. The PixiJS library is used for drawing graphics, which is also utilized in WebTigerPython. Section 3.3 will go in-depth on how PixiJS is used. For executing the Python code, the library uses Pyodide.

The key aspect that drew me to this library was the fact that it used both PixiJS and Pyodide, which are also used in WebTigerPython. The repository has stopped being maintained, but it provided insight into how arbitrary Python code could be executed in the browser. Just like with Pyxel, the problem with Enpyre was that,

```
def show():
    buf = BytesIO()
    matplotlib.pyplot.savefig(buf, format="png")
    buf.seek(0)
    img = base64.b64encode(buf.read()).decode("utf-8")
    matplotlib.pyplot.clf()
    self.output("img", img, contentType="img/png;base64")

matplotlib.pyplot.show = show
```

Listing 2.1. `override_matplotlib` in Papyros

since the authors had no reason to split their execution into multiple threads, they did all their processing in the main thread. This means the game window would freeze if the game logic were slow. That didn't work for WebTigerPython since we decided to run it in a separate thread to avoid window-freezing.

2.5 Papyros

Papyros [10] builds on a similar design as WebTigerPython. It is a public code editor for learning and executing Python code in the browser. The key features of it are its ability to interrupt code while it is running, a debugger, and being able to render and show Matplotlib [17]. Matplotlib is a Python library for creating visualizations and graphs. The ability to render Matplotlib in the browser was a potential candidate for being very similar to rendering GPanel on the browser canvas.

Papyros has a public repository allowing all the source code to be viewed in detail. Papyros renders Matplotlib libraries in a unique way. Pyodide is used to execute the Python code in a separate web worker to free up the main thread for a functional website interface.

The main thread transfers the Python code to the web worker to be executed in the background. All the Matplotlib functionality works without modification. The different part is the `plt.show()` function. In Papyros, this has been overwritten as seen in listing 2.1. Instead of trying to render the plot to the screen or canvas, it saves the image into a buffer, encodes it to base64, and then sends the base64 string as a message to the main thread. The main thread then renders the image at the correct location.

This method results in a significant overhead for any graphs created, which works for drawing diagrams but is infeasible for drawing more involved graphics and animations. Execution takes a few seconds to complete.

2.6 Via.js

When working with Pyodide and PixiJS in a web worker we initially struggled trying to find ways to pass objects from the main thread to the web worker and back. Some problematic objects included the `document` or `window` JavaScript object, which couldn't be passed as a message using Pyodide because only copyable objects could be passed as messages. Via.js seemed to be the solution at first glance. The library

documentation states the DOM[1] can be used in web workers and the main thread simultaneously [14].

Instead of somehow managing to copy the logic and objects across threads, it simulated the logic by keeping track of objects and callbacks on the receiver and receiving side. When importing a non-copyable, and hence non-passable, object from the main thread to the web worker, it assigns an ID and keeps track of the object in the main thread. Any callbacks added to the object would then be stored in a map with the respective ID. This way, the correct functions could be executed upon any action. This inspired the approach taken in Section 5.2.

[1] Document Object Model

Chapter 3

Tech Stack

WebTigerPython is the third iteration of the TigerJython application with the initial work done by Clemens Bachmann [2]. To better understand the approaches tested and the final implementation choices taken in the end, this chapter explains the core underlying technologies used.

3.1 Vue.js

Vue.js is a JavaScript framework for building web UI interfaces [39]. The frontend of WebTigerPython has been developed using Vue.js. The framework allows for the creation of a dynamic and modern website. For WebTigerPython, it enables a website where the canvas can be swapped out dynamically, and the Python print statements can be added next to the canvas without having to refresh the page.

3.2 Pyodide

TigerJython is an educational editor in which students can write and execute Python code [20]. Since WebTigerPython runs in the browser, there's no way to run Python straight out of the box. For that reason, Pyodide is used. Pyodide is a browser library based on WebAssembly, allowing for Python code to be executed in the browser [32]. This lets the code written by students in the editor run in their browser without having a separate backend executing the code or trans-compiling the Python code to another language like JavaScript.

WebTigerPython requires the Python code to be executed in a separate web worker to allow operations to be executed without freezing up the main browser thread. More information on why this is the case is given in Section 5.2. To make this step more accessible, a helper library called pyodide-worker-runner is used [15]. This library does the heavy lifting in managing Python libraries and passing messages between the main thread and web worker.

3.3 PixiJS

An HTML canvas is used for the best drawing performance in a browser. A library that gives a lot of control for drawing while helping abstract logic is PixiJS [12].

PixiJS is a JavaScript graphics library with over 40 000 stars on GitHub. It has various functions for drawing shapes on a canvas in an optimized fashion. Concepts like line width, basic shapes like rectangles and arcs, or the fill color, which existed in GPanel, also already exist in PixiJS, facilitating the development.

PixiJS was already used for the GTurtle implementation, so the goal was to stick to the same library for implementing GPanel as well [3]. For GTurtle, the main PixiJS library was used. This library only works in the main thread. This meant that any functions intended to be executed had to be executed in the main thread.

Chapter 4

Implementation

In this chapter, we go over how the final GPanel implementation was implemented in detail.

4.1 Web Worker

WebTigerPython handles most of its logic inside a web worker. Running code in a web worker results in the browser executing it in a separate thread. A single thread approach was tested, as explained in Section 5.1, but came with its complications. Using a web worker allows user-written Python code and all the logic around it to be executed in a separate thread without blocking the main thread. The main thread is responsible for handling the website itself and if the main thread is to be blocked, the main website becomes unresponsive.

4.2 PixiJS Web Worker Library

The final solution builds on the PixiJS web worker library [13]. It is a recent library to implement web worker support for version 7 of PixiJS and is not yet mentioned anywhere in the official documentation. This library is a subset of the main PixiJS library, which only exports the functions that work in a web worker [40]. This library made it possible to execute PixiJS drawing functions without having to send messages to the main thread with the dispatch action system. The next question was how to execute the PixiJS functions from Python code.

4.3 Pyodide

Standard browser functionality does not include the ability to execute Python code. One way Python code can be executed inside the browser is using Pyodide [32]. Using Pyodide, the PixiJS library is loaded into Python, allowing all PixiJS functions to be called inside Python.

When setting up Pyodide and the GPanel library in the browser, the canvas will be first sent from the main thread to the web worker as an `OffscreenCanvas` [4]. This allows the web worker to draw on the canvas while having the browser manage it so the drawn graphics can be seen in the browser.

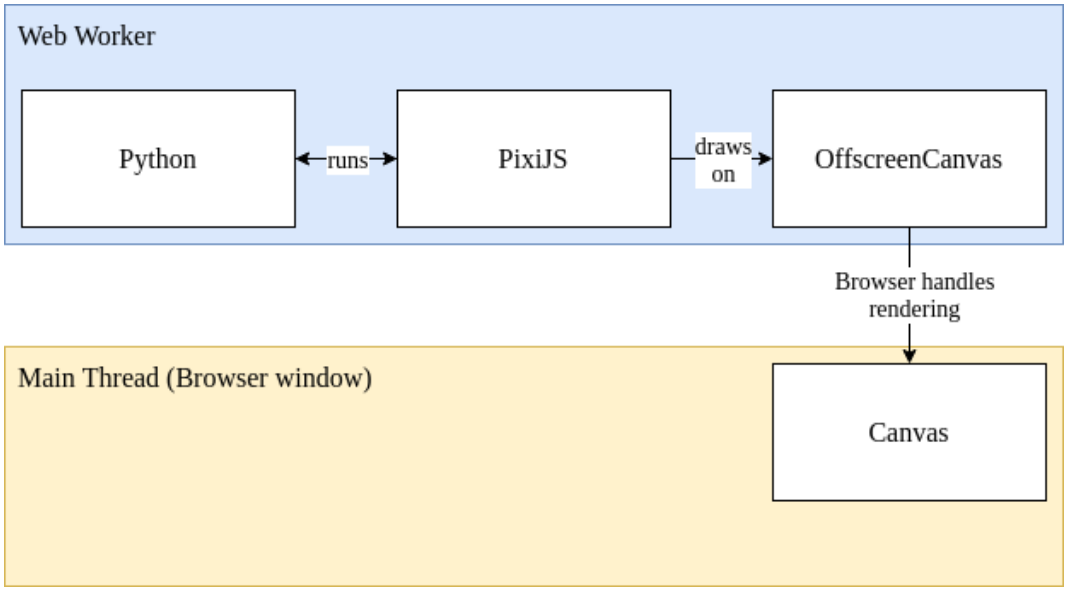

Figure 4.1. OffscreenCanvas allows the web worker to draw on the canvas.

Initially, when running GPanel code, a PixiJS application object is created with the given `OffscreenCanvas`. This application object stays persistent across every consecutive run, avoiding the need to initialize it again. For drawing on the canvas, PixiJS graphics objects are created and drawn with.

When executing a JavaScript function using Pyodide, Pyodide wraps the given function in a `setTimeout(function, 0)` call. This JavaScript function enqueues the callback on the callback queue to be executed after the given delay. In Pyodide's case, the delay is set to zero, which will run the function as soon as the event loop can execute its next task. The reasoning behind this is that Pyodide was initially made to be used in the main thread. To avoid blocking the main thread, the authors decided to off-put any JavaScript functions to be executed asynchronously [33]. In our case, when using Pyodide in a web worker, this is undesirable, though. The event loop is only capable of executing a single task at a given point in time, and the Python execution is considered a task as well. All JavaScript functions are placed in the callback queue to be executed later, often only once the Python code has finished executing.

Figure 4.2 shows a short code snippet that executes JavaScript code using Pyodide. All PixiJS functions will be called with `setTimeout(function, 0)` by Pyodide. This results in the function being placed on the callback queue to be called once the current executing Python task finishes and the even loop can start a new task. `print` and `console.log` both print a string to the console, but since `console.log` is a JavaScript function it will be placed on the callback queue to be executed later. The given example will print "Hello" before "World", even though the program order would indicate otherwise.

In the case of long-running Python code, JavaScript functions will be executed much later than initially intended. The best solution would be to wait for a JavaScript function to complete before continuing in Python, but as explained in Section 5.4, this was problematic to implement.

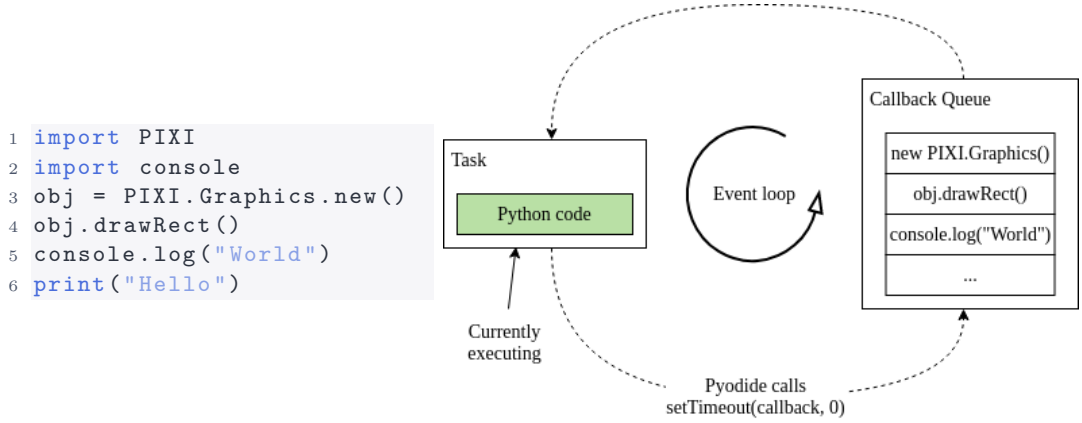

Figure 4.2. How the event loop looks when executing JavaScript functions using Pyodide in Python.

By periodically executing '`await asyncio.sleep(0)`' we can force the event loop to execute the next task on the callback queue [29]. The function has the negative side effect of being asynchronous. In Python, an asynchronous function can only be called in other asynchronous functions. For that, we had to find a solution for enabling the execution of '`await asyncio.sleep(0)`' without students having to know about or use asynchronous functions.

4.4 Concrete Syntax Tree

One main challenge was figuring out how to allow asynchronous execution in any user-written Python function. The final solution settled on was to modify the submitted code before executing it. Python code passed in was modified to turn any synchronous functions into asynchronous functions while wrapping any function calls with a call to an asynchronous wrapper function, which correctly executes any function synchronously or asynchronously.

The code was adjusted by editing the concrete syntax tree of the code. Every code element and formatting element is parsed and tracked in a concrete syntax tree. It allows moving through code as a tree and quickly editing function calls or definition nodes.

The difference to an abstract syntax tree is that converting a piece of code with comments to a concrete syntax tree and then back results in the same code. Converting that code to an abstract syntax tree and then back results in comments and unnecessary whitespaces being removed. It was essential to retain whitespaces and exact formatting so any errors in the modified code could be traced back and properly shown in the user-written code.

LibCST

Standard Python does not have an implementation for a concrete syntax tree. For that, LibCST was used. LibCST guarantees that turning a parsed code from tree format back into code is the same as the original code byte for byte by not only keeping track of managing abstract nodes like functions and arguments but also keeping

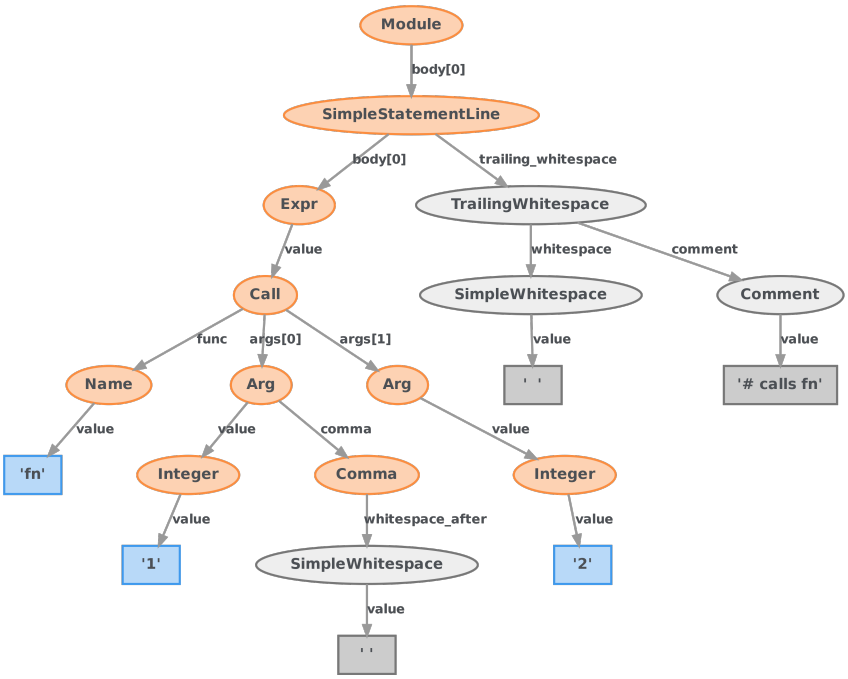

Figure 4.3. Sample LibCST concrete syntax tree built from the line '`fn(1, 2) # calls fn`'. *Image taken from the LibCST library documentation [22].*

track of parentheses and whitespace [21]. Figure 4.3 shows a sample tree generated by LibCST for the single Python line of '`fn(1, 2) # calls fn`'. Important to note are the nodes `Module` and `Call`, as those are the ones that get modified with the parser for this thesis.

Wrapper Function

Using the CST of the user-written code, it was possible to turn every function into an asynchronous function definition. To allow students to call functions as usual, any synchronous function calls that were executed had to be updated to execute the modified asynchronous function.

To this end, a general solution was used, which consisted of creating a wrapper function that checks during runtime if a given function is synchronous or asynchronous and then executes the function without or with `await`, respectively. This was the idea behind the `is_async` function. Every `Call` node could be modified using the CST library to then execute the `is_async` function under the hood.

listing 4.1 and listing 4.2 show how code with a function definition and call written by a user is modified. The `is_async` function then checks at runtime if a function is asynchronous or not using the `inspect.iscoroutinefunction` from the Python standard library [30].

Asynchronous Sleeping

The `is_async` function also keeps track of the amount of functions that have been called so that it can execute '`await asyncio.sleep(0)`' when required. Executing such an asynchronous sleep every few function calls enables PixiJS functions, which

```
def foo(x):
    ...
foo(1)
```

Listing 4.1. Example code a student could write.

```
from runner import is_async
async def foo(x):
    ...
await is_async(foo, 1)
```

Listing 4.2. Example code after being modified.

have been placed on the callback queue (see Section 4.3), to be executed and drawn on the canvas. This is the case even for a delay value of zero [29]. `asyncio` is a standard Python library for asynchronous execution. The `asyncio.sleep` function, in contrast to the `time.sleep` function, does not block the thread while sleeping and allows other functions to be executed.

A right balance has to be made between executing '`await asyncio.sleep(0)`' after every render function so that the canvas is instantly updated once something is drawn and executing it only for every defined amount of render calls to avoid big performance hits. Pyodide converts the call to `await asyncio.sleep(0)` into `setTimeout(callback, 0)` in its event loop implementation. It can result in a lot of performance loss when executing it since `setTimeout` can have a minimum delay of 4ms [5]. There is no fix in the current Pyodide version of 23.02 [36].

Initial GPanel implementations involved executing `await asyncio.sleep(0)` after every render call. This was a considerable performance dip and, on average, half the speed of the "Dispatch Action" method from Section 5.2. In the final implementation, the value of 100 was decided on for the number of functions that must be called for the program to sleep. This allows for the 4ms delay to occur only every 100 function calls, which is neglectable performance-wise.

Transformer

To work with the CST a transformer class is created. The transformer takes a concrete syntax tree and goes through it recursively node by node. Only three types of nodes are adjusted for GPanel's use case: The module, function definition, and function call nodes.

The module node is the root node of a program and contains a list of every statement line. This node is adjusted by adding an import to the `is_async` function at the top of the code to be used later.

To allow for every function to call asynchronous functions, every function definition is turned into an asynchronous function if it is not one yet.

Lastly, to support calling the functions made asynchronous, an `await` and the `is_async` function have to be wrapped around every function call.

listing 4.3 shows a code snippet as a student would write it. It is legal code because Python ignores the surplus whitespaces in the function header and the linebreak in the call to `foo`. listing 4.4 shows how the code looks after being converted to the asynchronous equivalent using the CST transformer. It is important to note that all the whitespace is kept intact, allowing for the display of traceback errors of the correct line. If for example, a student forgets to close a string in an argument to a function, the error will only be caught when executing the modified code. With the formatting staying the same, we can easily display the exact line where the error

```
def foo(x    ):
    ...
foo(
5)
```

Listing 4.3. Example code a student could write.

```
from runner import is_async
async def foo(x    ):
    ...
await is_async(foo,
5)
```

Listing 4.4. Example code after being modified.

```
def foo():
    return 0
for i in range(100_000):
    foo()
```

Listing 4.5. Benchmark for testing the performance of wrapping every function.

showed up.

Fixing Errors

When there is any error, including syntax errors, in the code, the error will only be detected once the code is executed, after the code has been modified. Python will then give a line number and a snippet of the line with the wrong code. The line number and wrong line snippet are also shown in the browser for the user.

The line number of an error can be decremented by one to circumvent the line taken up by the import statement since, with the concrete syntax tree, all other lines match one-to-one.

The code snippet will show the modified code if the error occurs in a function call. For example, `await is_async(foo)` will be displayed instead of simply `foo()` as the user wrote it. This can be quite confusing.

Before showing the exact code that resulted in an error, the error is piped through a function called `replace_is_async_string`. This function uses regex to clear out any mentions of `await is_async` in the error traceback.

Performance

To test the performance for wrapping every function in `is_async`, a simple benchmark was designed to test the execution speed for a synchronous function that returns a single value. listing 4.5 is the code used to test performance. The goal was to execute a function that does nothing to see how much overhead is caused by wrapping it in an `is_async` call.

Table 4.1 shows the average time taken to execute a single function call. The performance drop from the original to the modified version is, on average, around a factor of 12.6. This was a vital trade-off. Yet, there is a lot of space for improvement in terms of speed by circumventing this method.

Table 4.1. Comparison of average time to execute a function synchronously and modified to run asynchronously with $N = 100$. Benchmark ran on an Intel i7-8565U @ 4.6GHz with Python 3.10.

Execution method	Average time in ns
Original	81
Modified with async	1022

Limitations

A limitation of this approach is that it is not entirely foolproof, and there are ways to break it. The `is_async` function is appended with a randomized ID to avoid potential naming conflicts. However, a way to break the current implementation is by executing functions without calling them. One way to do this in Python is by using the `@property` decorator on a class method. This allows the execution of functions when accessing an attribute [28]. This results in the class method being turned into an asynchronous function because of the CST parsing. At the same time, the call to the property is not being awaited since, by parsing the CST, it is not possible to detect if accessing an attribute results in a function being called.

A second limitation concerning classes is that the Python function `__init__` cannot be asynchronous. Python returns an error if this function is defined asynchronously.

To at least allow for objects to be created, the decision was made to ignore the init function completely when transforming the given Python code. This has the side-effect that since the init function is synchronous and all other defined functions have been turned into asynchronous functions, it is impossible to call any other user-defined function in the init function.

Synchronous imported or built-in functions can be executed nonetheless because the whole init function scope is unmodified, and no function call is turned into an `is_async` call. Most GPanel functions are asynchronous and cannot be used in the constructor.

4.5 Graphics

The way graphics are drawn is split into multiple sub-parts of how the main drawing logic is handled in GPanel, as explained in the following section.

GPanel

GPanel functions are all implemented at the top level of `gpanel.py`. All the logic for reading in the right amount and correct types of arguments for a function is handled here. Every GPanel function has the same structure. It first checks if the GPanel object has been initialized using `makeGPanel`. Then, it counts how many arguments are given. If the arguments are supposed to be numbers or coordinates, they are validated to indeed be numbers. Once all the checks have passed, the corresponding function from the GPanel core is executed.

The GPanel core is where the main functionality and logic of all functions are handled. When executing `makeGPanel`, the basic idea is that we initialize a PixiJS

application in the background. This only happens on the first run since the Python context is kept between runs. On consequent runs, the old application object is completely reset.

The complete list of functions in the core is given in what follows. Any functions used by the public GPanel build on top of these functions or a combination of them. Some functions are also simply an alias of other existing functions. An example would be the `pos` function, an alias to the `move` function.

The central core functionality can be separated into five overarching categories. The *technical* category contains functions that are used to manage the GPanel instance in some form. These functions do not draw anything new on the canvas but instead manage what is already drawn on it or set corresponding options.

- Technical
  - `render`:
  - `flatten`
  - `window`
  - `erase`
  - `setBgColor`
  - `setColor`
  - `setLineWidth`
  - `move`
  - `getPixel`
- Non-fillable graphical
  - `line`
  - `draw`
  - `quadraticBezier`
  - `cubicBezier`
  - `point`
- Fillable graphical
  - `circle`
  - `ellipse`
  - `rectangle`
  - `polygon`
- Advanced graphical
  - `arc`
  - `fill`
  - `text`

Figure 4.4. Comparison of drawing an ellipse versus scaling a circle.

- Events
  - `addEvent`
  - `isKeyPressed`
  - `getKey`
  - `getKeyWait`

The *non-fillable graphical* category contains functions for drawing graphics that cannot be filled with color. Transferring these to PixiJS was a relatively simple process because PixiJS supports drawing pixels, straight lines, and quadratic and cubic Bézier curves out of the box.

The *fillable graphical* category contains functions that can be used to draw the outlines of a shape or draw the shape filled in with a color. These were also relatively simple to implement because PixiJS supports any of these shapes out of the box. A slight workaround had to be made for the circle. A circle will be an ellipse for different x and y-axis scales. When trying to draw a circle outline using PixiJS and adjusting the scale, the width of lines will also be adjusted, resulting in an uneven outline. Therefore, the circle function draws an ellipse under the hood, allowing for a consistent line width even with a stretched circle. Figure 4.4 shows the difference of drawing a scaled circle as an ellipse with a consistent linewidth versus drawing a circle and then scaling it.

The *advanced graphical* category contains functions that were more involved to implement. The arc function takes a radius, start angle, and extend angle to draw the part of the circle with the given radius from the start for the given extend angle amount of degrees. PixiJS supports an arc function but draws the arc, assuming that the xy-scale will be $1:1$. This results in a similar problem to outlining a circle. If the outline of an arc is drawn and then scaled, the outline will have an uneven width because it also gets scaled. To counteract that, the arc is approximated using Bézier curves [18, 31]. Using Bézier curves, the arc can be scaled arbitrarily while keeping the line width consistent. The method only works for circle ranges up to 90°, so up to four curves are drawn for higher ranges. Using the PixiJS fill feature, the curves can also be filled in for a filled arc.

The GPanel fill function is a flood fill, which takes a coordinate and a color as input. All colors equal to the starting coordinate that are touching will be replaced with the given color. PixiJS does not have any functions for filling by flood fill. This had to be implemented manually. The naive approach of flood filling using breadth-first-search in Python turned out to be too slow by taking multiple seconds to fill anything above 10 000 pixels (100 sq pixels). Python is 3–5 times slower in Pyodide [37]. Combined with the fact that Python is, on average, slower than JavaScript [11], I decided to implement the flood fill function on the JavaScript side and have the whole function be called on the Python side. This resulted in a

speed-up of around 3 but still meant that more extensive fill operations took multiple seconds. The final solution was to use a different algorithm to find the next edge and use PixiJS to outline the area to be filled using a border tracing algorithm [23]. Then, I could rely on the PixiJS feature, to draw lines that are filled in with a color. This resulted in an immense performance speed-up.

The text function allows a given string to be drawn to any coordinate. The font, text color, and background color can be adjusted as desired. Drawing the text is a built-in feature from PixiJS. PixiJS supports adding font styles, including whether the text should be bold or italic, what font family it should use, and more. As of now, any web font that is also supported by the browser can be used.

Image Flattening

PixiJS has problems rendering a large number of graphics. The main problem is that graphics objects are bulky objects that, if too many are created, use up a lot of memory and slow down the execution. As a result, only one graphics object is made and kept track of to be used for drawing shapes. This is also not without its flaws. The graphics object also becomes slower to operate with the increasing number of shapes drawn since the object keeps track of all lines and vertices internally. For this reason, the given variable `graphicsLimit` in the core `GPanel` sets a limit, indicating after how many drawn shapes the graphics object should be flattened into a single sprite object. PixiJS sprite objects are more lightweight, which allows the performance to be increased again every time the graphics object is flattened into a sprite object and the previous graphics object is reset.

An important consideration that had to be made was to add the contents of the graphics object to a single texture object. A previous attempt consisted of creating a new sprite object every flattens execution. This resulted in the number of sprites continuously increasing and the browser tab crashing because it used too much memory.

To test the performance of flattening versus not flattening I measured the average time taken to draw 10 pixels over the span of drawing a total of 10 000 pixels. This test shows how with more pixels, and hence graphics drawn, drawing pixels becomes slower and slower without flattening. Figure 4.5 visualizes the results.[1] The high spikes are due to the garbage collectors from Python and JavaScript, resulting in a few pixel placements taking longer. The code used for the benchmark is given in listing 4.6. Without flattening, the time required to render a pixel increases linearly. Flattening every 100 pixels results in an average execution time of 61.75 ± 10.5ms for rendering 10 pixels. Without flattening, the average execution time is roughly 105.84 ± 38.4 ms.

4.6 Events

GPanel has the feature to listen to events and execute corresponding callbacks. It is unique in this aspect, because it allows for events to be triggered asynchronously without executing a while loop of sorts to check for new inputs.

[1] The graph shows a single run for each option. Multiple runs were executed, and the result was consistently approximately equal.

```
from gpanel import *
from random import random
from time import time

n = 1000
makeGPanel(graphicsLimit=100)

for i in range(n):
    start = time()
    for j in range(10):
        point(random(), random())
    print(i*10, 1000*(time()-start))
```

Listing 4.6. GPanel benchmark to test the time taken for drawing 10 pixels with and without graphics flattening.

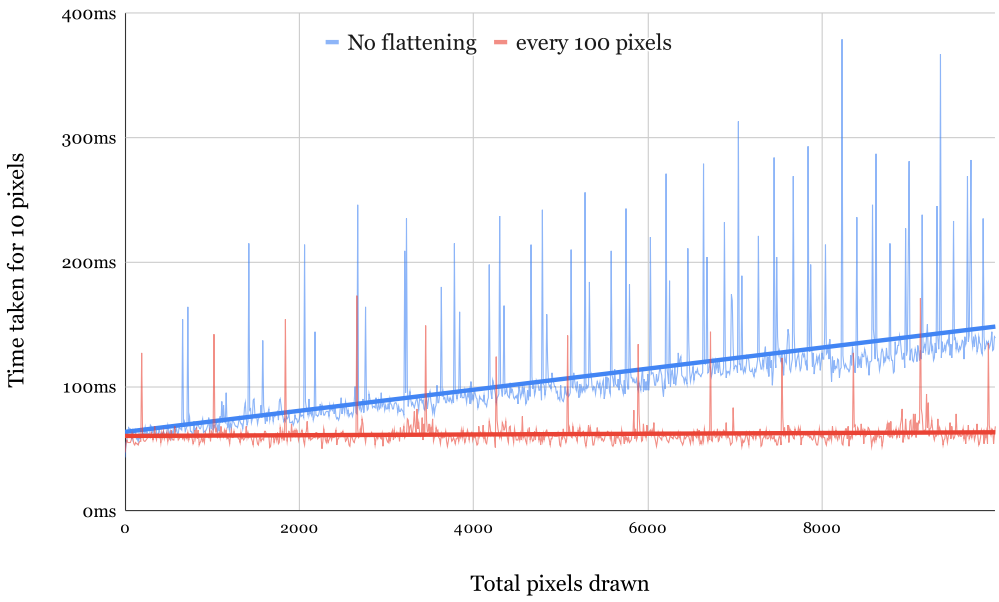

Figure 4.5. Comparison of time taken to draw 10 pixels with and without flattening. Benchmark ran on an Intel i7-8565U @ 4.6GHz with NVIDIA GeForce MX250 in Firefox.

Table 4.2. List of events supported by GPanel.

JavaScript Event	GPanel Event	Description
`keydown`	`keyPress`	When a key is pressed down.
`keyup`	`keyRelease`	When a key is released.
`pointerdown`	`mousePress`	When the mouse is pressed down or a tap is registered.
`pointerup`	`mouseRelease`	When the mouse is released or the finger stops touching the display.
`click`	`mouseClick`	When a full click or tap (down and up) is executed.
`dblclick`	`mouseDoubleClick`	When a double click or tap is executed.
`pointermove`	`mouseMove`	When the cursor is moved over the canvas.
`pointerleave`	`mouseEnter`	When the cursor enters the canvas.
`pointerenter`	`mouseExit`	When the cursor exits the canvas.
	`drag`	When the mouse is pressed down while being moved.

In the main thread, there are nine events that get listened to using the JavaScript function `addEventListener` [9] as shown in Table 4.2. These are then passed to the web worker. The drag event is special because the way it's used by GPanel is different from the drag event defined in JavaScript. The JavaScript drag event is triggered upon dragging an HTML element, while in GPanel, it is used for the notion of dragging the mouse.

Whenever an event is triggered, we send a message to the web worker using `postMessage` [7]. In the web worker, we also use the `addEventListener` function to listen to incoming messages and execute the corresponding callbacks for a given event as defined by the user. This allows us to execute callbacks asynchronously. Figure 4.6 shows this visually.

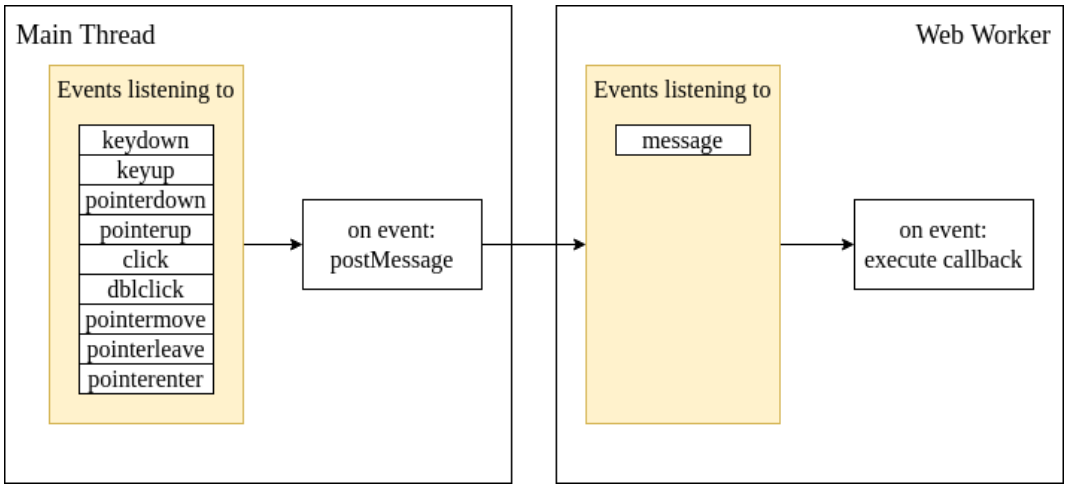

Figure 4.6. Representation of what events get listened to in the main thread and web worker.

Chapter 5

Other Approaches

The solution presented in this thesis was the result of trying out multiple approaches. Initially, the goal was to find an excellent way to pass messages and commands from the web worker to the main thread and back. Later, other problems showed up, resulting in more approaches that had to be tested. This section should also serve as a documentation source to see why specific ideas were not used in the final implementation.

A web worker-compatible PixiJS library was only discovered later, which is why many approaches depended on passing messages and commands between the web worker and the main thread.

5.1 Single Thread

The already existing GTurtle implementation runs in a web worker. A critical test that had to be made for my implementation for being able to use PixiJS was to test if it was possible to run WebTigerPython in a single thread and figure out if it would be possible to get it in a usable state this way. This would make a wide range of previously known problems easier. It would avoid passing messages between the main thread and the web worker. This in turn avoids a lot of overhead, improving performance and reducing the required code as there wouldn't have to be any logic for passing and handling messages in both the main thread and web worker.

For this experiment, I created a new website with only the bare requirements to better test for issues caused by using a single thread. This allowed me to focus on testing more specifically. The two essential requirements were having a code editor and a canvas where the drawing could take place. The code editor used the same library as in WebTigerPython called CodeMirror [16].

Figure 5.1 shows the test website, which was made for experimenting with running both the code editor, Python execution, and graphics drawing, all in the same main thread. It allowed a finer level of testing different Python programs without having any unrelated libraries interfering with the corresponding tests.

A simple program like the one given in the figure, which only drew a single rectangle, didn't result in any issues since it was instant. Problems arose with more involved programs that took a noticeable amount of time to execute. While those programs ran, the whole website was not usable anymore.

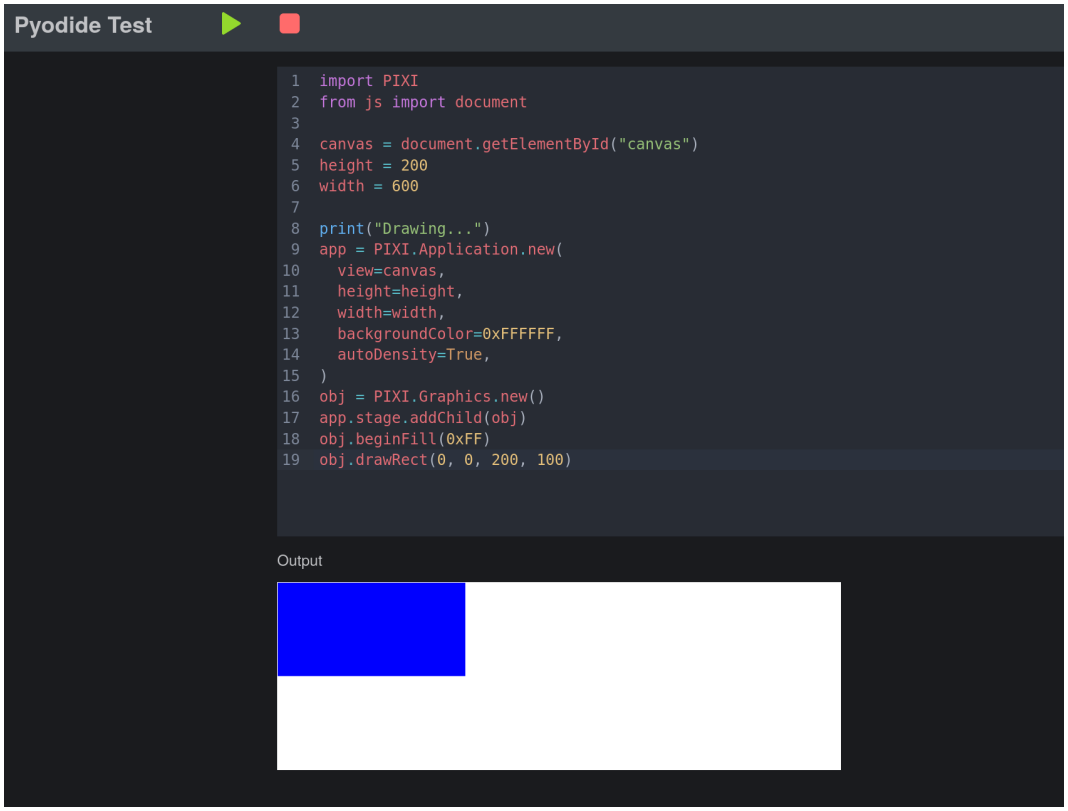

Figure 5.1. Test website to experiment with running Python code and drawing graphics with PixiJS in the main thread.

In a browser, the main thread manages the website state. All the logic regarding events and functionality takes place in the main thread. This means the website stops functioning if the main thread is stuck on executing something else, like long-running Python code. Every time long-running code was run, all the buttons on the website and entering code in the editor stopped working until the program finished.

There is no proper way around this other than to execute the logic and long-running Python code in a web worker. That frees up the main thread to handle browser events such as canceling the currently running program or even being able to click around the editor and edit the code.

5.2 Web Worker

After testing the single-thread approach, it was decided that the Python execution should occur in a web worker. This allowed for the page to continue functioning while the Python code, which was potentially slow, continued running in the background.

Two major approaches were looked into for working with a web worker. The first consisted of passing messages between the web worker and the main thread to execute functions. In contrast, the second method was based on a specific PixiJS web worker library that could be used to draw without passing any messages.

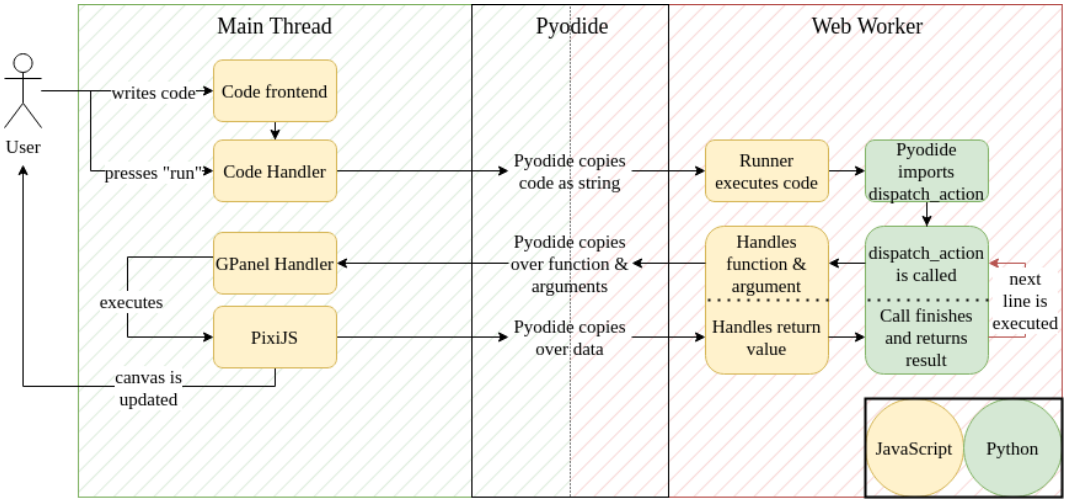

Figure 5.2. Execution flow for executing functions using dispatch_action.

```
def dispatch_action(class_name, fn_name, *args, **kwargs):
    data = {
        "className": class_name.value,
        "fnName": fn_name,
        "args": args,
        "kwargs": kwargs
    }
    res = defaultrunner.callback("pixi", data=data)
    return json.loads(res)
```

Listing 5.1. The `dispatch_action` function used to execute functions in the main thread.

Dispatch Action

The first approach was to dispatch actions from the web worker to the main thread. In this approach, every time a function was to be called in the web worker, the `dispatch_action` function, as seen in listing 5.1, was executed with the given class and method to execute with the required arguments. The class and method parameters were enumerated types, which allowed for type-checking the called function in Python to a certain degree without executing the code. The `pyodide-worker-runner` library made the passing of the message more organized. It enabled passing along a `type` string when executing the callback, which could be caught on the JavaScript side to better handle what should be executed. Any callbacks with the "`pixi`" type were then handled by a `handlePixi` function in the main thread, translating the executed function to the corresponding PixiJS function.

Figure 5.2 shows an overview of the execution flow when writing code and running it by a user of WebTigerPython. The middle part, depicted as Pyodide, is intermingled with pyodide-worker-runner, which includes managing and executing the Python code and passing messages from the web worker to the main thread and back.

Executing functions by passing the function and arguments from the web worker

```
from gpanel import *
from time import time

makeGPanel()

st = time()
for _ in range(100):
    point(0.1, 0.1)
print(time() - st)
```

Listing 5.2. Benchmark test for timing the drawing of 100 pixels.

Table 5.1. Comparison of time taken to draw 100 pixels with $N = 30$. Benchmark ran on an Intel i7-8565U @ 4.6GHz with NVIDIA GeForce MX250 in Firefox.

Execution method	**Average time in ms**
Dispatch Action	248 ± 48
PixiJS Native in Python	11 ± 3
Final GPanel Implementation	29 ± 4

to the main thread resulted in multiple drawbacks. For one, there was an enormous performance hit by transferring the messages back and forth. Another major drawback was that every function had to be coded twice. Once on the Python side with the logic for executing the function and once on the JavaScript side to receive the message and correctly execute the function with the given arguments. This became highly tedious when working with multiple objects and class methods.

listing 5.2 shows a snippet used for testing the performance of drawing 100 pixels. The test was restarted 30 times to ensure there were no problems with garbage collecting, as the "Dispatch Action" version did not correctly flatten multiple graphics, making long execution slower as time passed. More information is given in Section 4.5.

The average time to draw 100 pixels over 30 reruns can be seen in Table 5.1. There, we can see the comparison of drawing by passing every single function call between the web worker and main thread ("Dispatch Action"), using the PixiJS web worker library directly in the Python code ("PixiJS Native in Python"), and the final GPanel implementation. The final GPanel implementation also builds upon the PixiJS web worker library but adds another layer of logic, resulting in a slight performance loss. More on the PixiJS web worker implementation can be found in Section 4.2.

Object Map

One of the problems when going the "Dispatch Action" route was that passing certain objects using Pyodide did not work. Pyodide can be used in a web worker and uses the JavaScript native `Worker.postMessage()` function underneath [35] to pass objects back and forth. The `postMessage` function only allows sending a certain set of objects which can be copied according to the "structured clone algorithm" [7]. A major drawback is that it cannot copy and pass DOM nodes or, more generally, objects with functions. This meant that no PixiJS objects created in the main thread

```
    class Graphics:
        ...
        @property
        def width(self):
            response = dispatch_action(PixiClasses.graphics, "width")
            return response["width"]
```

Listing 5.3. How the Python property decorator was used to get values from the original object in the main thread.

could be passed and used in the web worker since they all have functions and are, hence, non-copyable. However, keeping track of the objects was important so that calling methods that modify object states could be called at any point.

The solution was to create an object map in the main thread. Every object created was added to the object map with an ID and then passed to the web worker. In the web worker, a wrapper object was created, which stored the ID of the original object. This resulted in an original JavaScript object and a Python wrapper object being created every time a new object was initialized. The ID could always be passed along with each request on every method involving that object so that the original object was modified or used. Since the JavaScript side never got refreshed, the map had to be cleared out every time the program was started to prevent it from resulting in a performance drop.

This added another layer of problems. Foremost, object attributes on the Python side had to also be methods. There was no reliable way of figuring out whether an object method changed any attributes or whether the value was copyable. Therefore, any object attributes on the Python side were masked functions, which sent a request to the main thread to receive the correct value. listing 5.3 demonstrates how this looked for the example `Graphics` class.

Code Generation

The next problem was that every incoming message had to be explicitly handled in the main thread to execute the correct function with the right arguments. For every supported function, there had to be code on the Python side, which allowed for calling the function, and then there had to be code on the JavaScript side, which was executed if the corresponding function name was passed as a message. This quickly resulted in a lot of manual work and many places for mistakes to slide in when adding more functions.

An approach towards making this easier was to use code generation to generate the proper handlers and entry point functions. For that, Jinja2 was used, which is a Python templating library. It allows the creation of templates, which can then be filled in with the correct arguments [24].

For WebTigerPython, creating a template file for the JavaScript handling part and a template file for how a class with its methods should look in the Python-PixiJS wrapper part was possible. With these two files, it was then possible to create an initial data file (in JSON or YAML) consisting of a list of classes, each with their name and attributes, and a list of methods, each with their name and arguments. This list could then be passed to Jinja2 with the template files to generate new

```
import PIXI

def drawGraphics():
    graphics = PIXI.Graphics.new()
    graphics.beginFill(0xFF0000)
    graphics.drawRect(0, 0, 100, 100)
    graphics.endFill()
```

Listing 5.4. Sample program to create a red 100-pixel wide square using PixiJS.

fully functioning files filled with all the required classes, all methods with their corresponding arguments, and handlers executing the suitable PixiJS functions.

This approach allowed for defining the general logic ahead in the templates, and then it was fast to create a lot of the required code for each function, as long as the information for the function was present. This was also one of the problems, though. There was no good way of collecting the required data. Two options that were considered included fetching the structure information from the PixiJS documentation website directly. At the same time, the other would have involved using the TypeScript typing and source code to figure out what functions to create wrappers for. Neither was a clean solution, so this approach was not a good way to go.

Another problem was in that each class had to be handled differently as they required different parameters to initialize. Initializing an application object, for example, required the canvas object to be passed as an argument. Some functions required other objects, and some had optional or an arbitrary number of arguments. With more classes and functions resulting in more edge cases, using code generation didn't make much sense. Adding on to that was that as the size of the templates grew, it became harder to read and understand them. For future maintenance purposes, this was not a clean approach.

Pyodide

Pyodide already has a lot of the required features. It allows objects and modules from JavaScript to be made accessible in Python with a name. In Python, these modules can be imported by the given name and used with only minimal adjustments in some syntax. Differences, for example, include having to call `Graphics.new()` in Python instead of the `new Graphics()` from JavaScript [34].

listing 5.4 shows a sample snippet of how PixiJS could be called in Python using Pyodide without having to write any additional bindings. Pyodide does a lot of the heavy lifting, making this approach exceptionally clean and reliable.

This approach has two significant limitations. The former being that calling functions with the wrong amount of arguments, or mistyped keyword arguments, does not result in an error, but instead causes the program to silently fail, while the latter is that the Python linter in code editors cannot deduce the types of the functions, making development more difficult. The developer will have to know exactly what arguments a function takes without any help from the editor.

In JavaScript, a typical pattern uses an object to pass arguments instead of simply a list of arguments. This allows optional arguments to be given in any order, which is

not supported in JavaScript. However, PixiJS does not throw an error if an unknown key value is defined. The PixiJS `Application` class takes an object as an argument of which one of the key values is called `view`. Constructing such an application object in JavaScript would take the syntax of `new Application({view=...})`. The equivalent in Python with Pyodide would be `Application.new(view=...)`. The problem arises if, for example, "view" is mistyped and instead spelled as "views". This does not throw an error, is very hard to spot, and results in something not working. In the case of the `view` key, it would result in nothing being drawn on the canvas at all.

This ties together with the Python linter not being able to deduce the types and consequently not being able to suggest errors before runtime. Pyodide does not support giving the needed information to the linter to figure out what modules and functions have been added with Pyodide. Using PixiJS in Python requires working closely with the documentation or playing around inside a TypeScript file to receive type hints for more complicated functions where documentation may be lacking or unclear before copying over the syntax to Python.

An option would be to write all PixiJS logic in JavaScript and then export functions to the Python side. However, the goal of the thesis was to offload as much as possible to the Python side, so it was decided that all the PixiJS logic would also be in Python. This means that you only have to work on and modify Python when maintaining GPanel in the future.

5.3 OffscreenCanvas

The next problem after getting the PixiJS web worker library working in the web worker was figuring out how to draw on the correct canvas. The canvas element on the web page is handled in the main thread and couldn't be passed over to the web worker using messages as it was not copyable (see Section 5.2). The solution for that was to use an offscreen canvas. An offscreen canvas allows for the rendering of a canvas element to be handled in a web worker [4]. The basic workflow is to create a canvas in the main thread, which can then be converted to an offscreen canvas. The offscreen canvas is a copyable object and can be sent to the web worker as a message. In the web worker, the offscreen canvas is then handled as a standard canvas and used to draw on. The browser will then correctly render what was drawn on the canvas on the web page.

This method allowed for using all the required PixiJS functionality and drawing on the canvas all in the web worker.

5.4 Overwriting Pyodide Calls

As explained in Section 4.3, something interesting happens when drawing on the canvas by calling PixiJS inside Python and having the code execution block because of slow code. The drawings only show up once the Python code finishes. The problem is that every JavaScript function in Python is executed using `setTimeout` in Pyodide with a delay of zero [33]. Executing `setTimeout` with a delay of zero results in the given function being placed at the end of the callback queue. When executing

```
def foo():
    def bar():
        ...
```

Listing 5.5. Original function definitions.

```
async def foo():
    async def bar():
        ...
```

Listing 5.6. Functions made asynchronous recursively.

a JavaScript function from PixiJS, we want it to have finished once we go to the following line in the Python code.

Editing the Pyodide Source Code

Two approaches were tried to circumvent the problem. The first idea was to edit how Pyodide handled the asynchronous event loop and overwrite it so that it executes functions synchronously instead of asynchronously. I tested editing the event loop implementation of Pyodide, but I could not get it to properly execute JavaScript functions synchronously. Ultimately, I decided against going this route, as it would require keeping track of a Pyodide fork or hot fixing the Python code in an unreliable way.

Modifying the Abstract Syntax Tree

As explained in Section 4.4, by calling `asyncio.sleep`, other tasks in the callback queue are able to be executed by the event loop. Adding an `await asyncio.sleep(0)` after every call to PixiJS allowed the event loop to switch contexts, execute the respective PixiJS function and draw on the canvas. This turned out to work quite well but came with its own set of problems.

The first problem was that to execute an asynchronous function in Python, the `await` keyword has to be used, and the function has to be inside an asynchronous scope. For example, in another asynchronous function. Because of this, simply adding `await asyncio.sleep(0)` after every PixiJS call did not work as it created invalid code.

The initial approach for this was to use the standard Python AST (abstract syntax trees) library [27] to go through the Python code written by a user and adjust the code so that every function definition was an asynchronous function definition. Later the AST library was exchanged for the CST library. This solution allowed for `await asyncio.sleep(0)` to be called without issue.

listing 5.5 shows two function definitions, with `bar` being defined in `foo`. listing 5.6 shows how the functions are gone through recursively and turned into asynchronous function definitions.

The next problem that had to be solved was that it was impossible to call these modified functions the usual way anymore. While the synchronous function `foo` from listing 5.5 can be called with `foo()` before it was modified, it had to be called with `await foo()` after being made asynchronous in listing 5.6, or else Python would throw an error and not execute the function. Because it cannot and should not be expected that users know about and can use asynchronous execution in Python, there had to be some automatized way of injecting `await` statements into the code.

```
    def drawGraphics():
        if input() == "1":
            return asyncFunction()
        else:
            return syncFunction()
```

Listing 5.7. Sample function for showcasing a conditional asynchronous function.

An initial idea was to manually go through every function and check whether it should be awaited or not. This turned out to be unreliable because there is a limit to how much context can be gathered from viewing the syntax on its own. listing 5.7 lists an example of a function that executes an asynchronous function conditionally. Just by the syntax alone, it is not possible to know if the function returned by `drawGraphics` should be awaited or not. A more general solution had to be found by using a generic wrapper function as explained in Section 4.4.

5.5 Handling events

GPanel allows for creating games and reacting to user events like key presses and mouse clicks. A way to handle events had to be figured out for this to work. The major challenge when working with user events is that in a web worker, there are only three events that can be listened for: `error`, `message`, and `messageerror` [8]. Any other event like `pointerdown` or `keypress` can only be listened for in the main thread. Therefore, all events were handled in the main thread and sent to the web worker as a message. There were multiple approaches to how events were to be stored and passed to the web worker. Additionally, a method had to be found for handling asynchronous events.

Event List

Initially, a similar approach to Pygame was tested. Pygame is a library for creating games in Python. In Pygame, key press events are placed in a queue. To check for key presses, the `pygame.event.get` function can be used [26]. The idea is that Pygame code is to be run synchronously, and the game loop, often a while-loop, continuously calls `pygame.event.get` to check for what key was pressed, ,and, if any, handles it accordingly. This approach works quite well, allowing for a more Pythonic way of "listening" for keyboard inputs instead of handling them asynchronously.

A similar approach was tried in WebTigerPython. In the main thread, a list of events was kept. Every time the mouse was clicked or a key was pressed, the event was added to the queue. Like in Pygame, a maximum length of 50 was set, so the oldest queue elements were deleted once there were too many. On the Python side, the dispatch action function resulted in the first element of the queue being popped and returned to the web worker to use. This worked for handling events synchronously when checking them on command. GPanel allows for asynchronous events by adding callbacks, though. These callbacks will be executed whenever one of the defined events occurs.

The first idea was to create a loop to be executed synchronously every time

```
async def event_sleep(self, delay):
    end = time.time() + delay
    while time.time() < end:
        event = self.getEvent()
        if event is not None:
            # handle executing the correct callback
```

Listing 5.8. General idea behind how the sleep function was modified to busy wait for events.

there was a slight delay, check for events, and run the corresponding callback. Sleep functions like GPanel's `delay` function were replaced with a busy wait type of function, which in a while loop kept checking for new events until the given delay was over. listing 5.8 shows an example of how the busy wait sleep function looked conceptually. To support callback functions being executed even after the user-written code completes, a while loop was added at the end of the program to handle events similarly.

Handling events in this fashion was unreliable. If no sleep function was called, and instead the program simply did a lot of operations by, for example, iterating over large lists, resulting in multiple seconds of execution, no event callback was being called until the end of the program execution. This was similar to how PixiJS functions were postponed until the end as explained in Section 4.3. Additionally, it was quite an impractical solution.

Pyodide Proxy

The second approach was not to keep track of a list of events in the main thread but instead, propagate all required events as messages to the web worker. In the web worker, messages can then be listened to, similar to how it would have been done in the main thread. This allowed for offloading a big part of the logic to the Python code, putting most of the logic in a single place. Using Pyodide, it was possible to add event listeners by calling `addEventListener("message", callback)` in Python whenever a callback was to be added [9].

This way the Python callback functions were executed by the JavaScript code of the browser asynchronously. This worked well together with the previous solutions of enabling asynchronous rendering. Since the JavaScript code ran independently of the Python code, it could also execute the callbacks after the Python code stopped. One slight adjustment that had to be made was to turn the given function into a Pyodide proxy using `create_proxy` to allow for it to be still callable even after the Python code has exited [34].

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis started out by testing different approaches for implementing a coordinate graphics library in WebTigerPython. The goal was to find the best solution that could be used to build GPanel in particular. Towards the end of the thesis, GPanel was additionally implemented, serving as a way to properly test the final approach taken.

To arrive at the final implementation, different ideas were tested. Initially, a single-thread solution was built as a prototype, which showed the need for keeping the Python execution in a separate web worker to allow for a usable website interface. Different libraries were looked into to figure out how to allow for a JavaScript library to be executed in Python. This included passing messages from the web worker to the main thread to allow for the latter to execute any drawing operations, or to execute drawing operations directly in the web worker, which brought along additional issues with asynchronous code execution.

Handling user events also was a challenge to consider. The two main implementations tested were to keep a big part of the event logic in the main thread and busy wait until an event was executed. The second approach, using browser event listeners, allowed for a more clean implementation.

Finally, we opted for using a web worker compatible version of the PixiJS library which could be executed in Python using Pyodide. To allow for users to write asynchronous code, the user-written code was modified using a concrete syntax tree library. This approach also allowed listening to events asynchronously by using the browser event listeners.

6.2 Future Work

The GPanel implementation presented in this thesis is not fully finished. Some aspects have to be added in future iterations. Library dependencies like PixiJS and Pyodide are being updated independently in the meantime, which might be beneficial for making future implementation more performant and reliable.

PixiJS v8

The currently used release version of PixiJS is version 7. In this version, web worker support is given as a separate package (Section 4.2) since the main package cannot be used in the web worker. Version 8 brings web worker support to the main package [1], but that version has not been released and is not stable as of the writing of this thesis.

In the future, when version 8 is released, the PixiJS version in WebTigerPython should be updated so that a single package can be used instead of two separate ones. Feature-wise, it would not bring much to the table other than having a more organized set of dependencies and using a more well-tested web worker implementation.

Further Functions

A few functions present in the original TigerJython GPanel have been left out because of time constraints. There are three groups of functions which have not been implemented as part of this thesis which could be added in a future iteration:

- **UI functions** allow a user to create and display components and show and update status texts. PixiJS would allow for creating interactive components on the canvas, making these features implementable, but it was infeasible to implement these functions for this thesis.

- **Storing graphics.** TigerJython allows for storing graphics and drawings so that they can be restored at a later point. These functions have not been implemented. An adequate way to implement such a storage system would be to use the browser's `localstorage` system, which allows storing data that persists across browser sessions [6].

- **Working with images.** TigerJython supports uploading images that can be placed at a given coordinate. Implementing a way to upload pictures to be used in WebTigerPython was out of scope and would require an update to the website UI.

Dynamic Canvas

One current problem is that the HTML canvas is set to a fixed size of 500 by 500 pixels. That results in the canvas being slightly covered by the code or barely visible if the window is too narrow. A future change should allow for dynamically adjusting the canvas size and correctly resizing the already drawn GPanel graphics. A potential challenge might include having to adjust the graphics from PixiJS. But it may be as straightforward as simply adjusting the scale attribute of the graphics and sprites currently on the canvas.

This change depends on how GPanel will eventually be fully implemented in WebTigerPython. For this to work properly and seamlessly with the GTurtle implementation, a second canvas has to be created, which can be swapped out since the GTurtle canvas is infinitely big while that from GPanel has a fixed size.

Pyodide Overwrite Webloop

One of the major problems faced in this thesis, as seen in Section 5.4, was that Pyodide executes imported JavaScript functions asynchronously by placing them on a separate queue. Many of the asynchronous workarounds made in the final implementation are because of that. If, in the future, Pyodide makes it easier to adjust the Pyodide Webloop implementation, this would result in a significant performance improvement by allowing everything to run synchronously as well as allowing for a lot of code to be cut out, making the code base easier to oversee and maintain.

Asynchronous Class Constructor

During the implementation of this thesis, classes turned out to cause more issues than initially anticipated. Classes are in quite a messy state because class methods are also adjusted when modifying user code. Still, an exception had to be made to the init function because the init function of a class cannot be asynchronous. This breaks any use case where a user wants to draw anything or call any other method in the init function.

In Section 6.2, we gave first ideas of how to fix this in the future. If it were possible to cut out the requirement to execute everything asynchronously, the init function and classes in general would work as intended.

Another issue that has not been explored a lot is that there might also be other in-built Python class methods that do not support being asynchronous apart from `__init__`.

Bibliography

[1] Support for v8 for PixiJS and @pixi/webworker package. https://github.com/pixijs/pixijs/issues/9946, 2023. Accessed 2024-02-15.

[2] Clemens Bachmann. WebTigerJython 3 A Web-Based Python IDE Supporting Educational Robotics. Master's thesis, ETH Zurich, 2023.

[3] Julia Bogdan. Turtle Graphics for WebTigerJython-3. Bachelor's thesis, ETH Zürich, 2023.

[4] Mozilla Contributors. OffscreenCanvas. https://developer.mozilla.org/en-US/docs/Web/API/OffscreenCanvas, 2023. Accessed 2024-02-15.

[5] Mozilla Contributors. setTimeout: Reasons for delays longer than specified. https://developer.mozilla.org/en-US/docs/Web/API/setTimeout#reasons_for_delays_longer_than_specified, 2023. Accessed 2024-02-15.

[6] Mozilla Contributors. Window: localstorage property. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage, 2023. Accessed 2024-02-17.

[7] Mozilla Contributors. Window: postMessage() method. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage, 2023. Accessed 2024-02-14.

[8] Mozilla Contributors. Worker - Web APIs: Events. https://developer.mozilla.org/en-US/docs/Web/API/Worker#events, 2023. Accessed 2024-02-16.

[9] Mozilla Contributors. EventTarget: addEventListener() method. https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener, 2024. Accessed 2024-02-16.

[10] Dodona. Papyros: Scratchpad for Python and JS, running in the browser. https://github.com/dodona-edu/papyros, 2021. Accessed on 2023-02-06.

[11] Isaac Gouy. Python 3 versus Node.js fastest performance. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python.html. Accessed 2024-02-17.

[12] Mat Groves and Chad Engler. PixiJS. https://pixijs.com, 2013. Accessed 2024-02-08.

[13] Mat Groves and Chad Engler. @pixi/webworker. https://www.npmjs.com/package/@pixi/webworker, 2022. Accessed 2024-02-15.

[14] Ashley Gullen. Pyxel. https://github.com/AshleyScirra/via.js, 2021. Accessed 2024-02-07.

[15] Alex Hall. pyodide-worker-runner. https://github.com/alexmojaki/pyodide-worker-runner, 2022. Accessed 2024-02-08.

[16] Marijn Haverbeke, Adrian Heine, et al. CodeMirror. https://codemirror.net, 2018. Accessed 2024-02-08.

[17] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[18] k88lawrence. Stack Overflow: How to best approximate a geometrical arc with a bezier curve. https://stackoverflow.com/questions/734076/how-to-best-approximate-a-geometrical-arc-with-a-bezier-curve, 2016. Accessed 2023-12-28.

[19] Takashi Kitao. Pyxel. https://github.com/kitao/pyxel, 2018. Accessed 2024-02-01.

[20] Tobias Kohn. *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. Doctoral thesis, ETH Zurich, Zürich, 2017.

[21] Meta. LibCST - A concrete syntax tree parser and serializer library for Python. https://libcst.readthedocs.io, 2023. Accessed 2024-02-16.

[22] Meta. Why LibCST? https://libcst.readthedocs.io/en/latest/why_libcst.html, 2023. Accessed 2024-02-16.

[23] The University of Iowa. Border tracing. https://user.engineering.uiowa.edu/~dip/LECTURE/Segmentation2.html. Accessed 2024-02-17.

[24] Pallets. Jinja2. https://jinja.palletsprojects.com/en/3.1.x/, 2007. Accessed 2024-02-14.

[25] Seymour Papert. Teaching Children to be Mathematicians Versus Teaching About Mathematics. *International Journal of Mathematical Education in Science and Technology*, 3(3):249–262, 1972.

[26] Pygame. pygame.event — pygame v2.6.0 documentation. https://www.pygame.org/docs/ref/event.html, 2016. Accessed 2024-02-16.

[27] Python Software Foundation. ast - Abstract Syntax Trees. https://docs.python.org/3/library/ast.html, 2024. Accessed 2024-02-15.

[28] Python Software Foundation. Built-in Functions: property. https://docs.python.org/3/library/functions.html#property, 2024. Accessed 2024-02-15.

[29] Python Software Foundation. Coroutines and Tasks: asyncio.sleep. https://docs.python.org/3/library/asyncio-task.html#asyncio.sleep, 2024. Accessed 2024-02-15.

[30] Python Software Foundation. Inspect - Inspect Live Objects. https://docs.python.org/3/library/inspect.html, 2024. Accessed 2024-02-15.

[31] Aleksas Riškus. Approximation of a cubic bezier curve by circular arcs and vice versa. *Information Technology and Control*, 35, 01 2006.

[32] The Pyodide Development Team. pyodide/pyodide. https://doi.org/10.5281/zenodo.7570138, August 2021.

[33] The Pyodide Development Team. pyodide.webloop. https://pyodide.org/en/0.23.2/usage/api/python-api/webloop.html#pyodide.webloop.WebLoop, 2021. Accessed 2024-02-15.

[34] The Pyodide Development Team. Type translations. https://pyodide.org/en/0.23.2/usage/type-conversions.html, 2021. Accessed 2024-02-15.

[35] The Pyodide Development Team. Using Pyodide in a web worker. https://pyodide.org/en/0.23.2/usage/webworker.html, 2021. Accessed 2024-02-14.

[36] The Pyodide Development Team. Better performing event loop. https://github.com/pyodide/pyodide/issues/4, 2023. Accessed 2024-02-15.

[37] The Pyodide Development Team. Improve performance of Python code in Pyodide. https://pyodide.org/en/0.23.2/project/roadmap.html#improve-performance-of-python-code-in-pyodide, 2023. Accessed 2024-02-17.

[38] Nicole Trachsler. Webtigerjython - a browser-based programming ide for education. Master thesis, ETH Zurich, Zurich, 2018.

[39] Evan You and Vue Contributors. Vue.js. https://vuejs.org, 2014. Accessed 2024-02-08.

[40] Tianlan Zhou. Add @pixi/webworker bundle. https://github.com/pixijs/pixijs/pull/8698, 2022. Accessed 2024-02-15.

Eigenständigkeitserklärung

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgeschlossen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und die Betreuerinnen der Arbeit.

Titel der Arbeit: Implementing GPanel: A Coordinate Graphics Library for WebTigerPython

Verfasst von: Mark Csurgay

Ich bestätige mit meiner Unterschrift

- Ich habe keine im Merkblatt "Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Zürich, 1. Oktober 2025