Concurrency for Creative Coding

Arnaud Loonstra
Graduation Thesis, August 18 2015
Media Technology MSc program, Leiden University
Thesis Advisors: Maarten H. Lamers, Wan J. Fokkink (VU University, Amsterdam)
arnaud@sphaero.org

Abstract

Programming for multiple processors is a challenging task. Approaches to program concurrently require a thorough understanding of the computer. Not all people who program possess this. However, as processors are not getting faster, everybody will need to program concurrently eventually. Creative Coding is the practice of programming for being expressive. In this research we propose an easy framework for Creative Coders to program concurrently based on a paradigm of interacting entities. The proposed framework is tested on a group of Creative Coders. The research confirms that concurrent programming is very challenging, that concurrent programs require a different design and that users find it easier to program using the proposed framework.

1. Introduction

Since around 2003 we are witnessing a halt in ever increasing clock speeds of processors (Figure 1). Before 2003 we could expect a new processor with doubled speed every two years. Processor manufacturers have resorted to creating processors containing multiple processors (multicore architecture) to fulfill the increasing performance demand.

Programmers are since then forced to program computers containing more than one processor. However, taking advantage of every processor in a computer is a challenging task[1]. Some computer scientists even think programming multiple processors (concurrent programming) is too complicated for humans[2].

While computer science has provided many approaches into concurrent programming they require a thorough understanding of the computer or they alienate from existing practices. For example a non-concurrent programming approach requires thinking about the sequence of operations. A concurrent programming approach adds to that thinking about all possible operation sequences and determining what should not happen in order to prevent concurrent conflicts. This requires a very different mental discipline coming from a non-concurrent practice.

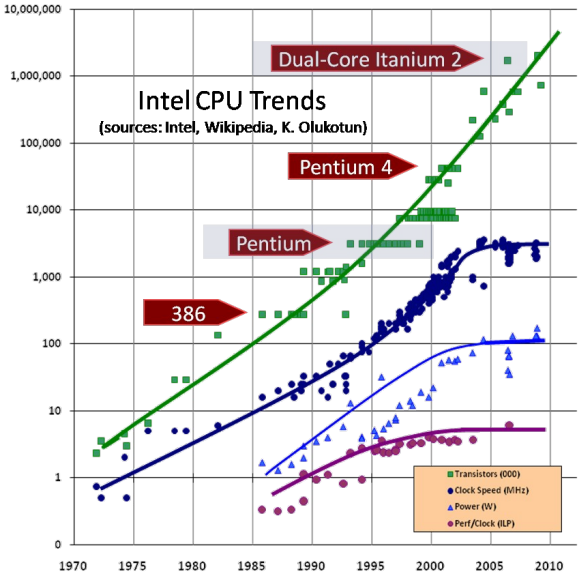


Figure 1: CPU trends. Courtesy of Herb Sutter. Source: gotwa.ca

Creative Coding is the practice of programming with the aim of being expressive rather than being functional[3]. Creative Coding emerged around the sixties when artists like Frieder Nake, Lillian Schwartz and Peter Struyken started creating and exhibiting art made with computers. The 1980's saw a lot of interest in the 'Demoscene' in which groups competed with each other to create the most technical and competent audiovisual creations. Contemporary Creative Coding can be found in many practices ranging from art to rapid prototyping. The scene is characterized by a strong

emphasis on making things work as opposed to elegance.

Commonly used Creative Coding toolkits such as 'Processing', 'OpenFrameworks' and 'PureData' are popular among artists and tinkerers. Such toolkits have an easy learning curve for programming. Processing, for example, was conceived specifically for teaching the fundamentals of computer programming in a visual context.

While Creative Coders do not necessarily possess the thorough understanding of the computer like computer scientists, they run into the same processor performance ceiling. Hence they are also required to explore programming multiple processors eventually. However, most Creative Coders are not up to the challenges brought by concurrent programming as they have no formal training in computer science.

Creative Coding projects are usually created from scratch. Therefore they can adopt new approaches very quickly as there is less burden from supporting legacy code. This makes this community an interesting target to research new paradigms brought by concurrent programming. In this research we are specifically looking for an approach to making all processors available to typical Creative Coders. For this we test a framework consisting of small sequential (non-concurrent) entities (Actors) which interact using a message passing model. The remainder of the paper discusses related works and then discusses an informal survey about the challenge of concurrent programming. We then propose a solution, discuss its implementation and our approach to testing it on Creative Coders. Finally we discuss the results and conclude with a discussion and future directions.

2. Related Works

Sutter's[4] article “The Free Lunch is Over” is an often cited article introducing the “fundamental turn towards concurrency in software” triggered by hitting the ceiling of processor clock speeds. Although Sutter hopes concurrent programming will become just as natural as object oriented programming, Lee[2] argues it never will. Lee argues that programming threads – a concurrent building block – discards the essential properties: understandability, predictability and determinism. Especially the nondeterministic property drives programmers into pruning every possible outcome of a program. He concludes his article by sending the engineering of threads into the engine room only to be touched by experts.

Herlihy & Shavit[5] address the issues related with concurrent programming to the fundamental limitations of the computational model. They deem it essential to acquire a basic understanding of concurrent computability. However, if the problems of concurrent programming root in the limitations of the computational model of programming, should we then stick with the computational model? Stein[6] argues that the computational model (metaphor) has become too dominant and addresses the need for a fundamental shift in the computational metaphor. The computational metaphor has enabled computer science to focus on the logical operations without worrying about the voltages inside the computer. By hiding physical processes of the computer we have seen enormous advances in computer science. However, with the advent of concurrency we are perhaps witnessing the limits of the traditional computational metaphor. Nowadays when we work at the computer many things are happening simultaneously. The sum of all these events are hard to explain using a traditional computational metaphor. Still sequential and centralized thinking associated with the computational metaphor are the dominant paradigm. Kolikant[7], referring to Resnick [8][9], mentions that our use of existing knowledge is responsible for the tendency towards centralized solutions and hence a sequential (computational) approach.

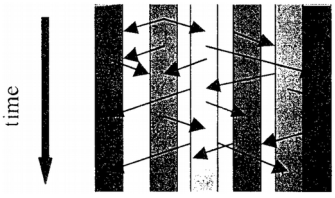


Figure 2: Computation as interactions paradigm as illustrated by L.A. Stein

This is expected to change as we are surrounded by more decentralized systems such as the Internet. Stein argues that "some of computation’s 'central dogma' ... blinds us to some of the truths of modern computer science". She proposes a 'computation as interactions' model in which entities communicate with each other. Her iconic model of this paradigm is depicted in figure 2. Computation does not reside in an entity but instead it is the result of the interactions amongst them. Stein also points out that for the students, including many females, who are uncomfortable with the rigid, linear and logical thinking of programming, an approach where partial programs are being pieced together might be more comfortable. “Componential tinkering may be precisely what is needed in today’s toolkit - and library-rich programming environment”. Stein's observation might fit typical Creative Coding practitioners just as well.

Both Sutter and Lee also talk about higher level programming models for concurrency. Lee proposes to focus on general purpose coordination languages instead of new languages. Lee refers to the Erlang language. Erlang is a language specifically designed for concurrency using a message passing paradigm. Exotic languages such as Erlang have not seen widespread adoption since they alienate from

existing practices. However, the ideas behind Erlang have found more adoption. Hintjens & Sustrik[10] talk about applying the lessons from Erlang to all programming languages. In a way they seem to adopt what Lee already argued for: coordination languages. The key in their argument is “to pass information as messages rather than shared state”.

A message passing paradigm is one of the three main concurrent paradigms described by Andrews & Schneider[11]. Message Passing is ubiquitous in concurrent programming and mostly used through the de facto MPI standard[12]. It seems a very intuitive paradigm as “synchronization is accomplished because a message can be received only after it has been sent”[11]. Platchetka[13] proposes a unifying framework using MPI and Petri nets for creating parallel applications. Using a visual environment (Kaira) users can rapidly prototype their ideas by manipulating program elements graphically. Platchetka's thesis discusses the Kaira environment in detail as well as other similar tools for visual programming. It is interesting to note that he did not find any explanation concerning the termination of other similar tools. Visual programming is known to have serious problems of visual representation, human perception and interpretation[14]. It leads to cumbersome and uninterpretable pictures which might be an explanation for their failures in common programming practices. However, within Creative Coding practices many visual programming environments, like PureData, have proved very successful.

Previous research by the main author has developed into a distributed computing framework[15]. This framework is similar to the Kaira environment but is an orchestration environment rather than a visual programming environment. This framework has been tested with Creative Coders. Results and concepts from this framework are used for this research and will be introduced further on.

3. An informal survey

3.1. Method

To test whether concurrency is used within Creative Coding practices and how it is regarded we have conducted an informal survey during the period of February till June 2015. The survey consisted of 9 score questions and 1 open question (Appendix A). The survey was announced on several Creative Coding community sites as well as social media. We specifically targeted Creative Coding communities using classical programming paradigms (ie, sequential programming, text-based, non-visual) as these approaches enable the programming of threads.

3.2. Results

In total 77 respondents filled in the survey. Of all respondents 49% considered themselves “Advanced” level programmer while 32% considered themselves “Intermediate” level.

Most Creative Coders (52%) have learned programming autodidactically. This is illustrative of the pragmatic approach which is associated with Creative Coding. People need something done which is nowadays often with the help of the computer. Hence this is how they get acquainted with programming.

With a few exceptions every respondent (88%) had experience with threads and agreed that threads are needed for their practice(87%).

The most frequent reason to use threads is to prevent a program from blocking (41%) followed by having the need for more performance (34%) and the need for low latency responses (25%).

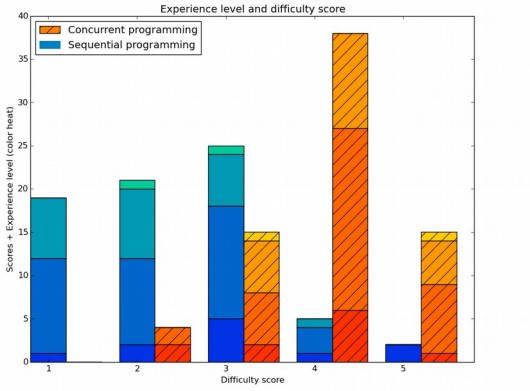


Figure 3: sequential and concurrent programming difficulty by experience level (color heat)

Figure 3 plots the experience level of the respondents with how difficult they consider sequential and concurrent programming. What can be clearly seen is that concurrent programming is considered much more difficult. In general this is regardless of their experience level.

Debugging when things go wrong is considered most difficult followed by mentally understanding what is happening and synchronizing access to data (figure 4).

The issues that are found difficult don't change much with increased experience level. A slight tendency towards debugging and away from the mental issues and random crashes can be observed.

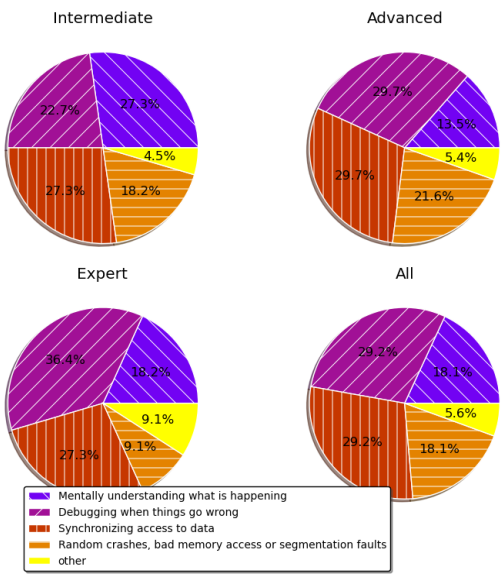


Figure 4: Most difficult concurrent topics split out by experience levels

3.3. Discussion

The conducted survey confirms the difficulty Creative Coders experience with concurrent programming. It indicates the expected issues the respondents experience but does not test whether these are the real issues. Since most respondents have already worked with threads we expect the respondents to be quite experienced Creative Coders. It is unlikely that Creative Coders will work with threads in the beginning.

Quite a few respondents remarked the need for a clear solution for utilizing multiple processors.

When the respondents were asked about how to deal with threads mentally some very diverse answers were given. It seems dealing with concurrent situations requires a mental model depending on the situation at hand. Often physical scenarios serve as a metaphor to explain the phenomena encountered, ie. parallel car highway lanes, boomerang throwing and catching, blind people building a house together. Some refer to computer science approaches, ie, mutexes, locks, different processes etc. The diverseness could illustrate the lack of a clear mental model which helps to understand the phenomenas encountered. This seems an important challenge for a framework which will be utilized by practitioners who might lack the thorough knowledge of a computer.

4. Proposing a solution

We take 3 approaches to utilizing all processors of a computer. The approaches share a message passing paradigm but differ in performance and context. All approaches relate to the 'computation as interactions' paradigm described by Stein[6]. Stein refers to entities in her model. In our proposal a process or a thread is regarded as an Actor. A network of Actors interact and together form the program. Actors exchange data by passing messages. An Actor sends data to a receiving Actor. The following three approaches are proposed:

1. Operating System process Actors
2. Actors as threads
3. Work distributed Actors

The proposed solution is specifically designed to separate the programming of a single actor from the programming of all Actors combined. Programming a single actor is a regular classic programming practice. The interconnections and communications of all Actors is programmed using a message passing approach.

Our first approach to utilize multiple processors in a computer is to run multiple processes. The scheduler of the operating system will divide all processes to all processors of a computer. This approach enables interacting entities of different programming languages as well as an easy integration of existing applications. However, it is limited to data exchange by copy as opposed to sharing data since the operating system fences the memory per program. For many use cases this approach is sufficient.

In cases where one does want to share data without copying one has to utilize multiple processors within a single process. This is accomplished by programming threads. Programming threads brings all the issues of concurrency described in the Related Works section. The second approach is therefore equal to the first but differs in the fact that data can be shared directly. There is no need to copy the data. A thread sends data to a receiving thread by a simple handover.

The third approach is one of a producer-consumer pattern. There might be situations where there is some work to be done for which we can utilize multiple workers. Following the paradigm of interacting Actors we clone a single Actor in order to create workers. The producer Actor sends a single set of work to each worker.

The three approaches cater for many situations a Creative Coder can run into. However, they are limited to the fact that data is passed before being operated on. This implies only one entity operates on the data. The operation an entity performs might be referred to as “embarrassingly parallel”, meaning the operation requires little or no communication. The opposite, inherently serial, is where operations completely depend on each other making any concurrent operation impossible. We can't expect our proposal to be a solution to all situations. However, we do wish to embrace any future demand. Therefore we need to assure our proposed solution embraces any existing and future solutions. This is

accomplished by the fact that programming an Actor relies on regular programming practices. The framework is merely a protocol between Actors and does not enforce a programming language nor paradigm when it comes to a single Actor. Thus a single Actor can use any concurrent approach it wants. This approach is a regular computational concurrent programming practice. We agree with Lee[2] that concurrent programming using a computational approach is very difficult. Therefore we consider this the engine room domain of our solution.

The solution proposed covers many situations which currently require a thorough understanding of programming threads and inner workings of the computer. We expect this approach to be well suitable for interactive programs and to offer an easy approach to utilizing multiple processors by combining small sequential Actors. For situations where more performance or custom logic is required the solution provides low level access using classic programming.

5. Implementation of the solution

To implement the proposed solution we build upon previous research[15]. Therefore we use the ZeroMQ library which provides us with a messaging framework. The ZeroMQ library is a high performance asynchronous messaging library which scored highest in a comparative transport libraries study by the CERN institute[16].

As the Actors in our solution need to communicate with each other we adopt the ZRE protocol[17] for discovery and data exchange. Although this protocol is aimed at discovery and exchange on a network we will enable this on inter thread communication by changing the message transports from TCP to Inproc[1].

The Actors are programmed according to an Actor Model[18] and running a Reactor Pattern[19] internally. The Reactor Pattern will use a poll() interface of the Operating System which enables us to listen on multiple events using a single blocking call.

Using the aforementioned building blocks we can cover the essential topics of the proposed solution. However, to enable dynamic, visual or live programming Actors need to exchange their properties and capabilities. This will be accomplished by using a meta data exchange protocol on top of the ZRE protocol. This meta data exchange protocol will be based upon previous research[15].

The implementation of the Producer-Consumer pattern can be accomplished using native interfaces provided by the ZeroMQ library. Since this research is focused on providing a framework for Creative Coders to utilize multiple processors rather than maximizing utilization this will be implemented and researched in a future study.

The framework features a tool to visualize interconnections of the Actors. We think it will be easier to visually see how Actors are related than to extract this from the source code of a program. As the framework uses a communication protocol, we can eavesdrop on this protocol to visualize the network of Actors. Figure 5 is a screenshot of the visualization tool showing 3 Actors. The visualization tool also supports editing the interconnections of the Actors.

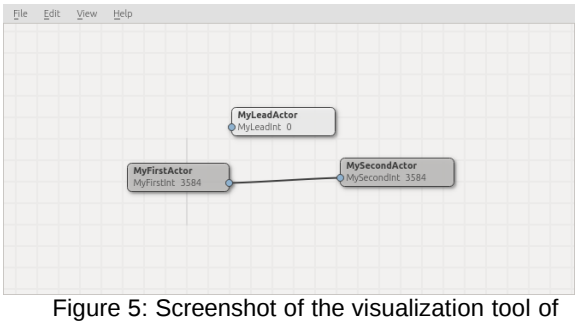


Figure 5: Screenshot of the visualization tool of the framework

The framework prototype is implemented using the Python programing language. Figure 6 shows example code of a very basic Actor in Python which increments a named variable (MyFirstInt) and emits it every update. The Actor class follows the application cycle of popular frameworks like Processing and OpenFrameworks through using setup(), update() and draw() methods.

```
Class MyFirstActor(Actor):

  def setup(self):
    self.register_int("MyFirstInt",0, "re")

  def update(self):
    self.emit_signal("MyFirstInt",\
                     self.get_value("MyFirstInt")+1)
```

Figure 6: Example Python source code of a basic Actor

The Python language is very well suitable for rapid prototyping as well as easing the path to lower level languages such as C. However, Python is limited by its Global Interpreter Lock to a single processor. Therefore it is impossible to utilize multiple processors using threads in Python. As this research is about the framework and its adoption by Creative Coders there is no need to utilize multiple processors. All challenges of concurrency still apply to Python as well. They are just limited to a single processor. The framework will be ported to other languages such as C once the adoption of the framework is satisfactory and provides a solution to the real world challenges of Creative Coders. Hence

1The in-process transport is a ZeroMQ transport passing messages directly via memory

the outcomes of this research are essential to test these questions and steer further development of the prototype and implementations in lower level languages.

6. Validating the solution

6.1. Method

To validate our proposal we test a prototype of the solution with a group of Creative Coders. The test subjects are introduced in the framework and need to program a solution using the provided framework for three provided scenarios.

The first test (figure 7) is an introduction to the framework and the behavior of it. After being instructed how to use the framework the test subjects are asked to create multiple painters which need to share a single canvas. Each painter is given a section of the canvas; however, only one painter can paint at a time. The canvas is divided in equal parts and each painter is assigned the section besides the previous painter. This first test is about resource sharing and workload division.

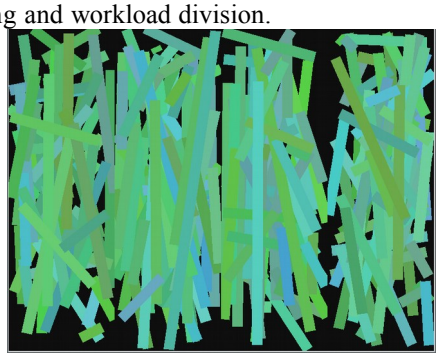


Figure 7: Screenshot of a test 1 application output running 4 random painting threads (Actors) and 1 thread to display

The second test (figure 8) is based on the "Dining Philosophers Problem". Five philosophers sit around a dining table. Each philosopher has a bowl of rice. When they eat they acquire food for thought. While they are thinking they philosophize about their thoughts. However, there are only 5 chopsticks which limits eating philosophers to 2 at the time. The philosophers need to acquire 2 chopsticks first before they can eat. In the second test the subjects need to prevent dead locks and starvation.

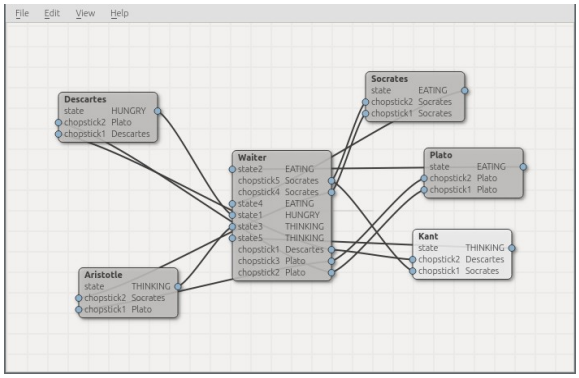


Figure 8: Screenshot of the visualization tool of a test 2 application using a waiter solution

In the third test the subjects are shown an example application in which the image of a webcam is displayed as well as thumbnails of processed images of the webcam (figure 9). The source code of the application doing one thumbnail process is given. The test subjects are asked to implement multiple thumbnails as done in the shown example. The subjects will then run into the issues of having multiple threads share a single image. As this is a difficult subject the test is devised mostly to let subjects understand the concept of sharing memory.

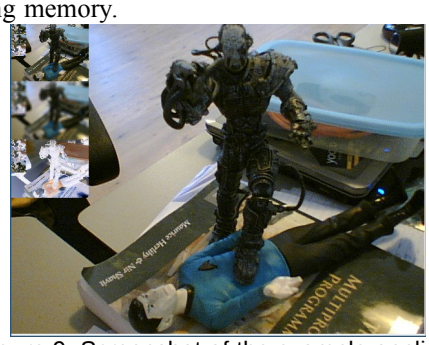


Figure 9: Screenshot of the example application output shown for test 3

For these tests we are interested in whether the test subjects understand the issues at hand, whether they are able to come up with a solution and how difficult they find building it using the provided framework. To measure this we acquire results by interviewing the test subjects and by a survey (Appendix B) similar to the informal survey described in section 3.

6.2. Results

Tests were done using 6 subjects all with a background in Creative Coding or tinkering with code. Subjects were selected for being familiar with text based programming, Python and the cited frameworks. There was a maximum of 7 test subjects due to available hardware. Unfortunately two subjects had to cancel. One replacement was found. The test subjects were each provided with a Raspberry Pi 2 machine which had the proposed framework and all necessary tools installed. Before doing the tests the subjects were introduced into concurrent programming and were given a workshop to use the proposed framework. During the tests the subjects could request assistance to clarify any issues they had with the framework or the tests. A full day was required to do the tests in which the test subjects simultaneously worked on the tests.

All subjects apart from subject 6 confirmed the conclusion from the informal survey in programming with threads being more difficult to sequential programming. Subject 6 regarded programming with or without threads equally difficult.

Subject 1 was unable to solve any tests. This was due to the fact that subject 1 was unfamiliar with Object Oriented Programming which the framework relies on. Therefore the test results of subject 1 are not reflected in the remainder of this paper.

Test 1 was solved by all subjects and they regarded the test with a difficulty index of 2 or 3. They understood the problem well and only had challenges in getting acquainted with the new framework.

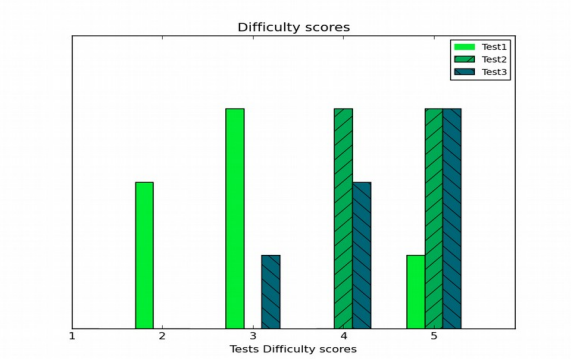


Figure 10: Tests difficulty scores

Test 2 was solved by subject 3. All other subjects needed more time to solve it as time was limited to around 3 hours. Subjects did express understanding the problem well and found designing a program for the test most challenging. Subjects regarded test 2 more difficult than test 1.

The final test 3 was solved by none of the subjects due to running out of time. Subjects regarded test 3 as less difficult then test 2 but more difficult than test 1. The concept of sharing memory was understood by the subjects as can be interpreted by their feedback.

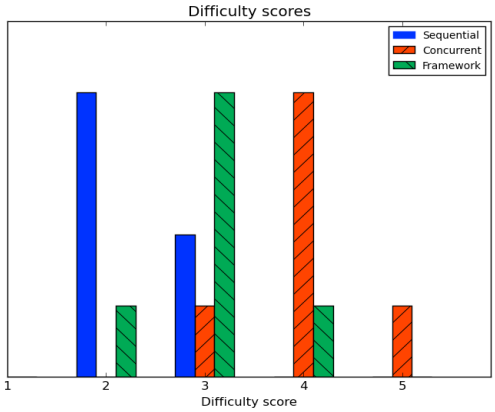


Figure 11: Programming difficulty scores from the user tests

Figure 10 plots how the test subjects rated the difficulty of the tests.

When asked to rate the difficulty of programming using the provided framework the subjects rated it more difficult than programming without threads but easier than programming with threads. (figure 11)

Four subjects answered positively when asked whether they could see themselves using the provided framework. Two subjects weren't sure.

For the tests documentation was written which the subjects could consult (Appendix C). This was written as a guide as well as typical programming API documentation. As the introductory workshop only covered the framework as a whole some finer details were noted in the documentation. We noticed many subjects only skim the documentation and move on to work on the assignment resulting in them making mistakes which were warned about in the documentation.

A typical example of where concurrency issues arise is when one accesses methods or members of a class which runs in a different thread than from where it is accessed from. While a program might often run without a problem it can run into very unexpected issues. It is very hard to find the cause of such issues. There is no way for the framework to prevent a user from creating these issues thus a user of the framework has to know about this. The framework can at most provide a conceptual model which prevents such issues. The message passing paradigm used in the framework provides an easy to understand conceptual model. The Actors (Threads) in the program need to communicate by passing messages not by directly accessing each others members. Once this was clear to the subjects they did not make such mistakes anymore.

The subjects requested more practical examples than the few examples given in the documentation. Some subjects preferred to work from an example which they can modify to their needs as was done in test 3.

Two subjects already had experience with the framework from the previous research[15] which dealt with distributed computing. These users had less difficulty with designing a program for the assignments as the design of the program is similar to designing multiple programs interacting through a network.

Every subject ran into three issues working on the assignments. They first needed to understand how they had to work with the framework. They then needed to wrap their head around the assignments. Finally they needed to design a program for the assignment. No subject complained about any concurrent challenges apart from the issue mentioned before. This is also reflected in figure 11 where the subjects find programming with the framework easier than concurrent programming.

Although the subjects don't have proven experience with concurrent programming they rated the difficulty similar to the difficulty ratings we found in the informal survey (figure 3).

7. Discussion/Conclusion

This research has started from the observation that it is very hard for Creative Coders, to utilize multiple processors. From our research we can confirm that concurrent programming is indeed challenging for Creative Coders. We have then set for an approach to simplify concurrent programming by providing a framework encompassing concurrent programming and an intuitive conceptual model.

We developed a framework based on current computer scientific models with the aim of providing Creative Coders with an intuitive conceptual model for concurrent programming. From our tests we observe test subjects being able to deal with the challenges given in the tests. They did not all succeed in creating a solution for every assignment, however, when asked, subjects expressed needing more time to finish their solution. We have found no reason to believe the subjects were unable or found it too difficult to design a solution.

The developed framework provided subjects with an approach into concurrent programming. It does not alter programming other than how to think about designing a program. Therefore subjects can run into all concurrency issues. However, as the framework provides a conceptual model which the user can follow it prevents many issues. The tests with the framework pose questions on how to make it more clear for the user to follow the concept of passing messages between threads instead of sharing variables. For example subjects easily made the mistake of referring to members of an Actor class from another Actor class. This could be due to the fact that in the tests all the Actor classes are declared in a single file. Perhaps it is less tempting if Actor classes were separated into individual files. It makes no difference for the program but perhaps it makes it clearer for the user. Another approach would be to not instantiate an Actor class but let the LeadActor instantiate Actors by passing the class type instead of an instantiation. This should be looked into further.

The current prototype of the framework does not provide any synchronization mechanisms. In test 3 users needed to do multiple operations on a webcam image. As there was no synchronization of the operations the operated images were slightly lagging the main image. This was no issue for test 3 but in real usage one would want to synchronize the Actors. This is especially needed when the producer-consumer approach of the proposed solution would be implemented.

We observed our test subjects to refer to a different approach into designing their program. This seems to illustrate the fundamental difference from sequential programming. To program concurrently one needs to think differently about elements of a program as Stein depicted in here 'computation as interactions' paradigm[6].

We also observed these changed design patterns being similar to the design of a distributed program (where a program consists of smaller programs running on different computers). Test subjects with experience in this found it easier to design a multithreaded application using the provided framework. We can expect this to also hold true for users familiar with creating applications for Internet as Kolikant[7] referred to. Design patterns emerging from a message passing paradigm are then applicable from distributed applications on the Internet to multiple cores of a computer.

As the framework relied on previous research[15] it uses a system of sensors and emitters which communicate through signals. This is a slight drift from pure message passing and causes slight extra overhead at startup and during Actor changes. The benefit of this approach was that users didn't need to think about what messages were passed. Just as with regular programming users only dealt with variables and when they needed to be communicated with another thread.

We believe the proposed solution is an easier approach into concurrent programming than regular (lower level) programming of threads using mutexes, semaphores etc. The proposed solution is an approach for Creative Coders to utilize multiple processors more easily while still leaving lower level options available to the Creative Coders. Using the proposed solution a Creative Coder can get acquainted with the design and behavior of a concurrent program. If one then requires better performance one can utilize lower level approaches. Therefore the proposed solution provides a sustainable path to more fundamental approaches into concurrent programming. It does not alienate from existing approaches and introduces its users into the design patterns of a concurrent program.

The results from the user tests suggest we were able to achieve our aims to provide easy concurrent programming to Creative Coders. To conclude we list the most important findings from this research:

- There is a problem for Creative Coders to utilize multiple processors without thorough knowledge of the computers
- Our proposed solution provides an easy and sustainable approach to utilizing multiple processors and designing concurrent programs

- The framework developed during this research, although not feature complete, supports a conceptual model which helps users to design concurrent programs

7.1. Future work

Besides implementing the missing functionalities, optimizations and found insights in the framework this research also has brought new questions. In this final section we discuss some possible future directions to continue this research.

A part of the developed framework consisted of an Actor Editor supporting the user in editing the interconnections and visualizing them. It is similar to common visual programming environments like PureData; however, it is solely meant to orchestrate and visualize the program. While visual programming is not uncommon in Creative Coding practices it is quite uncommon in more classical programming practices. We have noticed the Actor Editor being a very welcome tool to support understanding of inner program dependencies and even enabling live editing of the program.

The test subjects expressed wanting to be able to create Actors from the Actor Editor providing them a template Actor which they could then fill in through text based programming. It is therefore interesting to research this tool further as a tool which positions itself between classical text based programming and visual programming.

The visualization of the program also supports debugging efforts. Many Creative Coders do their debugging by using print statements. However, in a concurrent program this is not reliable as you cannot determine the order of the output from the print statements.

If a program would consist of many actors visualizing them becomes difficult. What approaches make visualization large interconnected Actors practical? Actors could be grouped together, filtering actors to focus only on the interesting parts. We could foresee visualizing a concurrent program becoming an important part of debugging the program.

As Actors communicate with each other we can talk about the topologies of how Actors are interconnected. We would like to research the lessons learned from networking infrastructures as they could apply to program topologies as well. Especially since the networking field is adopting software defined networking concepts.

Actors are aware of each other as they exchange meta data about each other. This is used to visualize the actors as well as edit them live in the Actor Editor. Through this approach Actors could also connect to each other autonomously. The meta data exchange protocol could be extended with logic so Actors could exchange without the developer expressing this before hand. Such an approach brings to mind scenarios applicable to Cellular Automata.

Programming concurrently using our proposed solution asks Creative Coders to think differently about the design of the program. What design patterns are required and how could such patterns be supported in the framework? Distributed Algorithms is another computer scientific field from which we can apply its lessons learned.

However, most important of all is to research adoption by Creative Coders and others. We have tested the proposal on a small group of Creative Coders. We have only touched the surface of the possibilities. Therefore development of this framework should need to keep a direct link with practical use cases. In previous research we have kept a tight link with practical use cases by letting developers, researchers and artists work together[20]. This assures the adoption by users is very visible and very verbose. This approach can be applied to any descendant research.

8. Acknowledgement

Many thanks to the participants of the user test who were willing to sacrifice a full day to this research. Also thanks to all the participants of the online survey. We are also very grateful to the MAPLAB of the Utrecht School of Arts and the z25.org Foundation for providing facilities for this research.

9. References

[1] P. Hijma, “Programming Many-Cores on Multiple Levels of Abstraction,” PhD Thesis, VU University, Amsterdam, 2015.

[2] E. A. Lee, “The problem with threads”, Computer, vol. 39, no. 5, pp. 33–42, May 2006.

[3] J. Maeda, “Creative Code. New York: Thames & Hudson, 2004.

[4] H. Sutter, “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”, Dr. Dobb’s Journal, vol. 30(3), Mar. 2005.

[5] M. Herlihy and N. Shavit, “The Art of Multiprocessor Programming. Morgan Kaufmann, 2012.

[6] L. A. Stein, “Challenging the Computational Metaphor: Implications for How We Think”, Cybernetics and Systems, vol. 30, no. 6, pp. 473–507, Aug. 1999.

[7] Y. B.-D. Kolikant, “Gardeners and Cinema Tickets: High School Students’ Preconceptions of Concurrency”, Computer Science

Education, vol. 11, no. 3, pp. 221–245, Sep. 2001.

[8] M. Resnick, “Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds”, Cambridge, MA, USA: MIT Press, 1994.

[9] M. Resnick, “Beyond the Centralized Mindset: Explorations in Massively-parallel Microworlds”, Massachusetts Institute of Technology, Cambridge, MA, USA, 1992.

[10] P. Hintjens and M. Sustrik, “Multithreading Magic”, 01-Sep-2010. [Online]. Available: http://zeromq.org/blog:multithreading-magic. [Accessed: 06-Apr-2015].

[11] G. R. Andrews and F. B. Schneider, “Concepts and Notations for Concurrent Programming”, ACM Comput. Surv., vol. 15, no. 1, pp. 3–43, Mar. 1983.

[12] D. W. Walker, “The design of a standard message passing interface for distributed memory concurrent computers”, Parallel Computing, vol. 20, no. 4, pp. 657–673, Apr. 1994.

[13] T. Plachetka, “Unifying Framework for Message Passing”, in SOFSEM 2006: Theory and Practice of Computer Science, J. Wiedermann, G. Tel, J. Pokorný, M. Bieliková, and J. Štuller, Eds. Springer Berlin Heidelberg, 2006, pp. 451–460.

[14] V. Averbukh and M. Bakhterev, “The analysis of visual parallel programming languages” ACSIJ, vol. 2, no. 3, pp. 126–131, Jul. 2013.

[15] A. Loonstra, “Orchestrating computer systems, a research into a new protocol”, FOSDEM 2015 conference, Brussels, 01-Feb-2015. [Online]. Available: https://fosdem.org/2015/schedule/event/deviot02/. [Accessed: 08-Apr-2015].

[16] A. Dworak, M. Sobczak, F. Ehm, W. Sliwinski, and P. Charrue, “Middleware Trends And Market Leaders 2011”, 01-Oct-2011. [Online]. Available: http://cds.cern.ch/record/1391410. [Accessed: 25-Apr-2015].

[17] P. Hintjens, “ZeroMQ Realtime Exchange Protocol.” [Online]. Available: http://rfc.zeromq.org/spec:36.

[18] M. Odersky and M. Odersky, “Scala actors: Unifying thread-based and event-based programming”, Theor. Comput. Sci, 2009.

[19] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, vol. 2. Wiley, 2000.

[20] A. Loonstra, “Freaklabs: Joint Artists and Developers Technology Design and Evaluation,” Stichting z25.org. [Online]. Available: http://www.z25.org/static/_rd_/freaklab_plab/index.html. [Accessed: 31-Jul-2015].

10. Appendices

10.1 Appendix A: Informal Survey

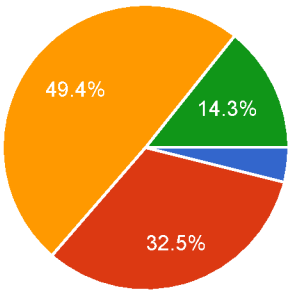
To test whether concurrency is used within Creative Coding practices and how it is regarded we have conducted an informal survey during the period of February till June 2015. The survey consisted of 9 score questions and 1 open question. The survey was announced on several Creative Coding community sites as well as social media. We specifically targeted Creative Coding communities using classical programming paradigms (ie, sequential programming, non-visual) as these paradigms enable the programming of threads.

77 responses

View all responses Publish analytics

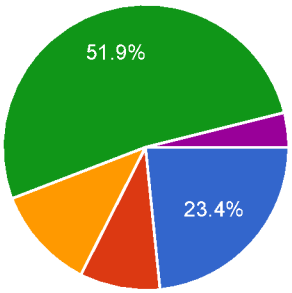
Summary

1: What level would you consider yourself as a programmer



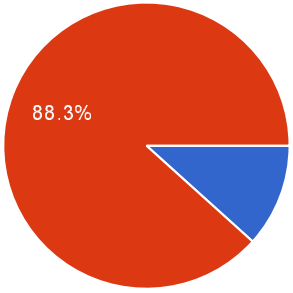
Novice	**3**	3.9%
Intermediate	**25**	32.5%
Advanced	**38**	49.4%
Expert	**11**	14.3%

2: How did you learn how to program



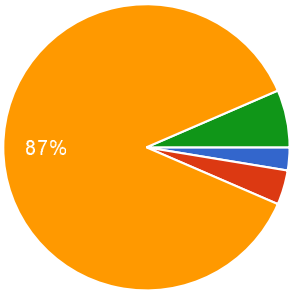
I studied computer science	**18**	23.4%
I was taught in school	**7**	9.1%
I followed a course and took it from there	**9**	11.7%
Autodidact, I taught myself	**40**	51.9%
Other	**3**	3.9%

3: Have you ever used threads in programming



No, I haven't	**9**	11.7%
Yes, I have	**68**	88.3%
I have no idea	**0**	0%

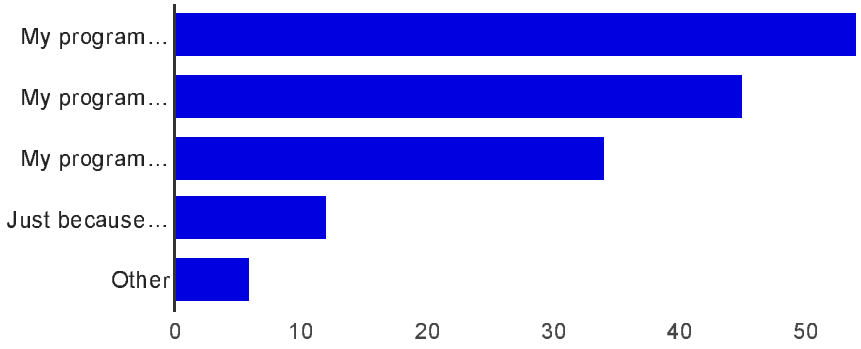
4: Are threads needed for your practice?



As I don't know what they are I can't say really	**2**	2.6%
No, I haven't needed them	**3**	3.9%
Yes	**67**	87%
I'm not sure	**5**	6.5%

5: Programming threads

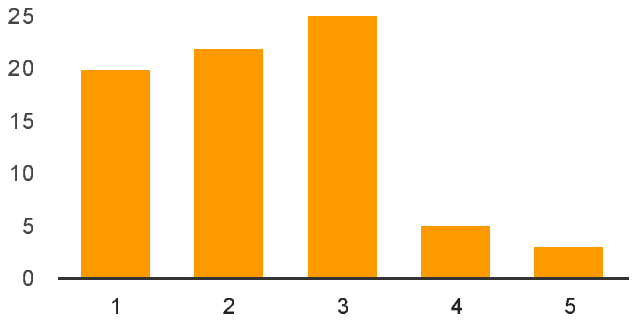
Why have you needed threads?



My program would otherwise block or my program had to wait for an answer	**54**	72%
My program was too slow and I needed better performance	**45**	60%

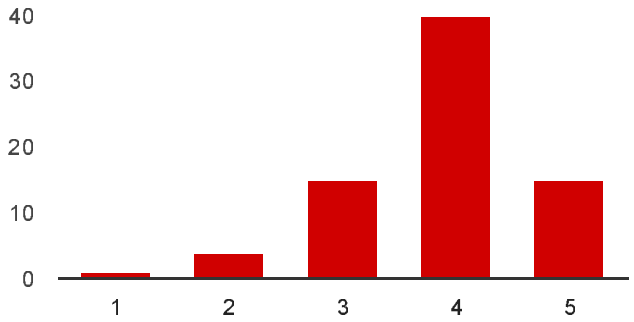
My program needed to response with as little delay as possible	**34**	45.3%
Just because I could	**12**	16%
Other	**6**	8%

6: On a scale from 1 to 5 how difficult do you consider programming *without* threads?



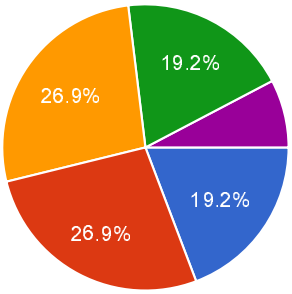
Very easy:	1	**20**	26.7%
	2	**22**	29.3%
	3	**25**	33.3%
	4	**5**	6.7%
Very difficult:	5	**3**	4%

7: On a scale from 1 to 5 how difficult do you consider programming *with* threads?



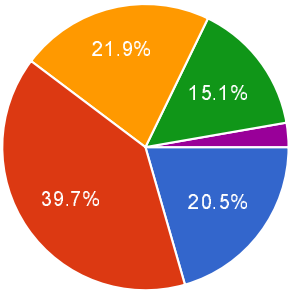
Very easy:	1	**1**	1.3%
	2	**4**	5.3%
	3	**15**	20%
	4	**40**	53.3%
Very difficult:	5	**15**	20%

8: What do you find most difficult about programming with threads?



Random crashes, bad memory access or segmentation faults	**15**	20%
Debugging when things go wrong	**21**	28%
Synchronizing access to data	**21**	28%
Mentally understanding what is happening	**15**	20%
Other	**6**	8%

9: What do you find second most difficult about programming with threads?



Random crashes, bad memory access or segmentation faults	**15**	20.5%
Debugging when things go wrong	**29**	39.7%
Synchronizing access to data	**16**	21.9%
Mentally understanding what is happening	**11**	15.1%
Other	**2**	2.7%

If possible, can you give a short insight of how you deal with threads mentally?

Send dirreferent tasks to different threads??

different processes running in parrallel that sometimes need to wait for eachother to share resources or do something with the main thread (like creating open gl textures)

As little as possible. I do not use them often and when I do it's mostly for getting some data from the internet without blocking the UI thread. I (heavily) rely on examples I find on the internet and hope that works :P

depends on the workload, sometimes it is simple enough to use openMP and

parallelize a loop, other times there are separate tasks that can be run in parallel or in a parallel pipelined approach with producer and consumer threads.

When possible I try to structure of a stack of (independent) jobs that need to be executed (hopefully on independent sets of data). The description of different processes running simultaneously matches up best with the way I try to use threads. i.e.: make stack of tasks ' jobs[]' while(waiting for next serial task to complete): foreach (thread in threads[]): if (not thread.working()) then thread.do(jobs[].pop())

Most of my threading requires synchronizing access to data, so I tend to think in terms of what resources need to be made available when and how.

If loops are objects moving circularly, then threads are the ones moving faster or slower, sometimes you adjust their speed to match the others.

I see it as another application "linked" to the main one, with a dedicated memory-management. It could be used to divide the charge of computation used in the rendering application, to be rendered as smooth as possible.

Concurrent processes that are not necessarily synchronized. Multiple processes working with common data.

I try to picture what if more than 1 Thread gets to access some variable at the same time. Is gonna be any serious consequence in that case and I gotta make sure everything happens atomically. Or it doesn't matter in that case and I let it be, since a slightly delay in getting the values updated won't cause any chaos.

different processes running simultaneously

I don't like when software blocks so I use worker threads. By using multiple threads i can run many tasks at once and utilize all the cpus. I keep the threads seperate and don't let them share memory (so I don't get confused) except for progress varables. For code that needs to run fast I use of. In areas where speed is important i expect it to be multithreaded if possible. I want to work on my idea not implementation and definitely spend as little time as possible oon optimization.

I always try to find ways to create lock free producer - consumer solution for my threads to get max performance.

I scribble together a graphic representation of what is going on.

I think about different process independent from the main

I think there great, just put things into arrays to take them from thread to thread.

I've found the consumer/producer pattern to be most useful in dealing with threading

Multiple processes accessing data concurrently, with atomic actions

For me it's mainly about splitting interface related logic from backend related logic, e.g. the handling of gestures from the user are handled on the main thread but making api call and parsing data is dispatched over different threads.

In many cases I like to have the best performance possible, so I often want a seperate process for handling the visual, the logic and the specific routine. The visual part can be any drawing, 3d or otherwise graphical routine. The logic being the way the program or game works. And the specific could be something like AI or another

resource intensive routine.

Mentally I see threads as 'multiple people performing baskets of tasks at the same time' instead of the non-threaded 'one person doing all the work sequentially'. As a mental construct this is quite an easy way to understand it, however in practice things can become quite complicated because these tasks (and their input/output) sometimes rely on each other.

I visualize them as stand-alone "functions" or "programs" that can take control of the CPU at any time, no matter how inconvenient, and may require access to data simultaneously with other threads. This ultimately means that data is never safe from harm. The more you learn about concurrency, the more you realize how non-trivial this is. I usually rely on concurrent FIFO buffers to pass data between threads. I've also worked with a task based approach, where 'work' is given to 'jobs'. I really liked this approach, as long as there is enough work to be done to keep the CPU cores busy.

I think about in the term of a highway with parallel lanes. A car can switch lanes, but cannot be at two lanes at the same time. Furthermore a car can only switch form lanes if there is enough "space" between the other cars.

approaching it as different processes running simultaneously

different processes

Processor or memory heavy stuff in a different thread. I mainly use it for performance reasons.

I guess I imagine that I'm forking off a new process/processes that run with a time independent of the main process.

For most of the work I have done I use simple lock_guards to block shared memory access in simple models. Usually just one extra thread to churn and produce new results. I usually just think of a thread as being a while loop that shouldn't touch any memory it doesn't explicitly own without a mutex lock. Lately I've been wrapping my head around the Windows idea of waitable objects and trying to have one thread signal the other to get moving again.

As different processes. I would think of it as something running simultaneously/parallel - but on a different core

swapping contexts

I do not use thread pools but I want to look into it. I use GLSL a lot even for GPGPU, e.g. encoding float data as colours and similar to retrieve computed data from a texture.

I generally err on the side of caution and protect everything, and then work backwards, removing unnecessary locks and whatnot

synchronising the data was mostly the hard part

I think of them as boomerangs you throw away at different speeds, Some thing you dont think about but you have to remember to catch when they(eventally) return.

" approaching it as different processes running simultaneously" is pretty much how I

think about it

For me it's about doing multiple processes simultaneously (even though in reality they might not be simultaneous). Or simply doing one main "thing", and something else on the background.

Two blind dudes building a house at the same time, nailing eachother in the foot by accident .

I imagine threads a small programs or processes sharing data and control. See see them in my near vicinity and up in the air connected in some way, running free and requesting access. It is a bit abstract. Metaphors can be dangerous, but I am interested in learning about a solid metaphor. Also different ones, each explaining a different ways concurrency.

threads = processing pipeline for specific input-output. ie - kinect input & processing happens in a thread and the result (blobs, person contours, etc) are then double buffered for acces in the main thread - main thread is opengl output thread; anything visual goes here So generally speaking I try to use as much threads as possible for input/output just so the main thread does not hang.

I think of threads as a separate space where things can happen. You pass them information you want them to operate on, and then later retrieve completed work. The interface to the threaded operations should be strictly controlled and tiny to prevent issues. I avoid working directly with threads, and instead use abstractions on top of them like futures (c++ std::future). I like how they provide a model of work to be done and a way to get the result of that work when it is finished.

In my practice where I'm mainly coding for systems that playback (generative) moving images on (large) screens, synchronising playback with the playback medium (screens/projection/LEDscreens) is of most importance. Every hiccup in the Imagestream will result in stutters or skipped frames in the screen output. Having a simple method to reserve resources strictly for the purpose of providing as smooth as can be playback (and at the same time controlling/checking the generation in a feedback loop, so you can know when things go wrong) would be a big, big help. Most CC /visual programming environments are more or less 'fire and forget' based: we just try to generate as many frames as we can and throw them at the screen, in daily practise, this leads to unwanted stutters and hiccups, even in large-venue systems that are used for big concerts and trade shows. On a side note: I'm not really sure if these issues can be solved on CPU level, it seems to me that the way we output image data out of our systems (mostly over HDMI) is interfering with smooth playback in a great deal too..

I think a lot about it but I don't come to an answer, sorry :(

I usually think in states. When running a piece of code that either uses or changes a state it is important that some other process cannot change that same state. It will have to wait it's turn until one piece of code is done so that each piece of code can handle all of it's logic, without changing states during. Also I sometimes like to think about is as 'event' which need to be handled, nut not at the same time.

I think of threads as separate processes running simultaneously. But I also keep in mind that memory can be shared among those two processes, so it is important to protect from deadlocks and race conditions. That is where things get hard to handle mentally.

If you have any suggestions or remarks regarding this survey please leave them below

I would love to see specific solutions outlined for common problems like: - computer vision on a separate thread to animation - preloading content in secondary thread, with a solution to generating open gl textures on the main thread

My main issue with threading is that there aren't very many good resources on how to implement threading patterns or good case studies. Most threading tutorials I've used are very basic and don't really extrapolate to real world cases very well, so I've learned most of the threading patterns that I use by reading other people's code and relying on standard reference to fill in the gaps.

Great initiative! Looking forward to hearing more about the research.

You should not need to use threads directly unless you are building a library for concurrent tasks (ex. an async dispatch system). In practice, you should be using a well tested toolkit that provides higher level async functionality with a well defined way to stay thread safe (ex. completion callbacks).

I would suggest to ask people for their profession, because I think it says a lot about the answers. Also was it hard for me to recognize this email was form you Arnoud, maybe you can improve that cause I almost ignored this survey.. - Robert-Hein (oud DVTG)

Questions 6 and 7 are a bit hard to answer if you don't provide any context. Programming an application that only shows an empty window is easy. Programming a single-threaded application that can load hundreds of images and still remains responsive is hard. Programming a multi-threaded image loader is easy. Programming the core classes for task based concurrency is hard. Good luck! Veel succes met je onderzoek!

I would add a textArea to precise the answers concerning the question 6 and 7

Great acknowledgement of a big problem in the creative coding community. Threads are still black magic to a lot of us because many don't have a traditional background.

Deja vu

share your results on the OF forum please :)

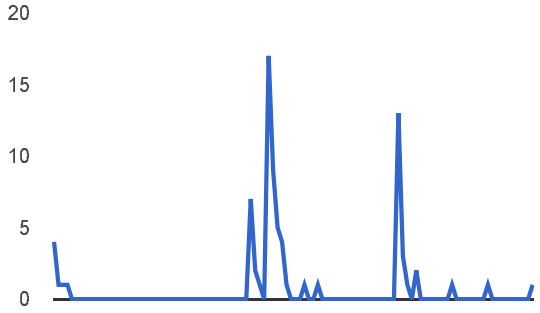
For me, it would be useful to have some examples of how to multithread useful alorithms, (i.e. feature detection in OpenCV) and possibly easy hooks to do these operations. Like with running different operations on an image, I believe there are different approaches. You can have a thread pool where each thread takes a whole image to process and buffer the results. Or with some operations you can split a single image into parts and reconstruct the output of each of these threads. I would

like to see examples of both. ^not sure I'm totally correct about that, but it's what I think I know :)

I used threads for flocking calculations to boost performance. I found implementing threads in processing/java easier than I had thought beforehand. And the goal of boosting performance was reached. However, the main problem I encountered was that other parts of the program were not as easy to adapt for concurrent threads and thus became new bottlenecks. I also looked into other parallel processing solutions such as OpenCL, but I found that much, much harder to implement and that it had many more restrictions than threads. I'm currently working in OpenFrameworks/C++, but I haven't tried threads in this yet. As stated before, I think a big part is not so much the difficulty of threads per se, but rather finding and implementing ways to use them practically, in particular because many resources (algorithms, code examples, etc.) are based on sequential solutions. A concrete examples being implementing PARALLEL k-D tree construction & range searches. Also see two of my Twitter conversations that are relevant here:
https://twitter.com/AmnonOwed/status/425729650401099776
https://twitter.com/AmnonOwed/status/425729902696886272

see above :)

Number of daily responses



10.2 Appendix B: User Test Survey

For the user tests we are interested in whether the test subjects understand the issues at hand, whether they are able to come up with a solution and how difficult they find building it using the provided framework. To measure this we acquire results by interviewing the test subjects and by a survey similar to the informal survey described in section 3.

7 responses

View all responses Publish analytics

Summary

1: What is your age

35

43

24

30

31

2: How many years have you been programming

3

5

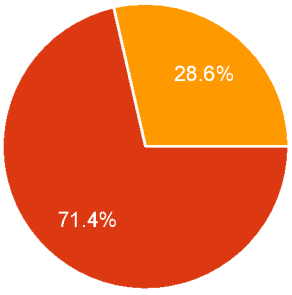
8

17

15

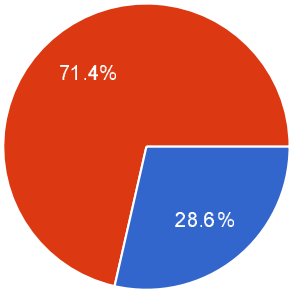
12

3: What level would you consider yourself as a programmer



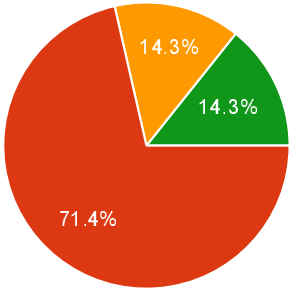
Novice	**0**	0%
Intermediate	**5**	71.4%
Advanced	**2**	28.6%
Expert	**0**	0%

4: Have you ever used threads in programming



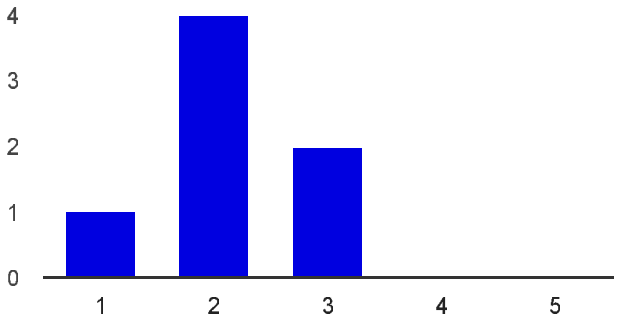
No, I haven't	**2**	28.6%
Yes, I have	**5**	71.4%
I have no idea	**0**	0%

5: What level would you consider yourself as a Python programmer



Novice	**0**	0%
Intermediate	**5**	71.4%
Advanced	**1**	14.3%
Expert	**1**	14.3%

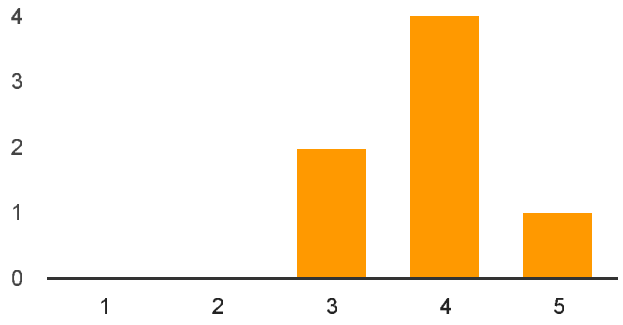
6: On a scale from 1 to 5 how difficult do you consider programming *without* threads?



Very easy: 1	**1**	14.3%
2	**4**	57.1%

3	**2**	28.6%
4	**0**	0%
Very difficult: 5	**0**	0%

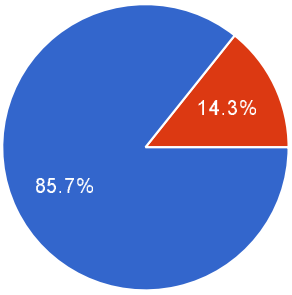
7: On a scale from 1 to 5 how difficult do you consider programming *with* threads?



Very easy: 1	**0**	0%
2	**0**	0%
3	**2**	28.6%
4	**4**	57.1%
Very difficult: 5	**1**	14.3%

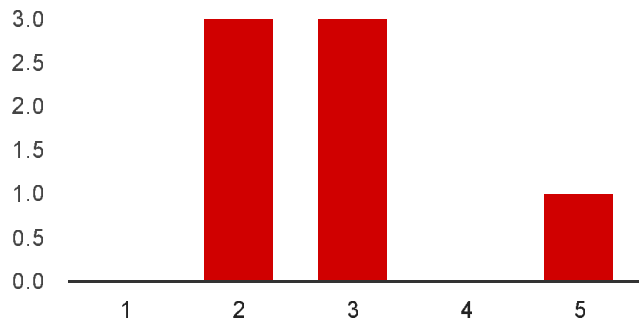
User tests

8: Were you able to solve test 1 (Painters Spree)?



Yes	**6**	85.7%
No	**1**	14.3%
If I had more time	**0**	0%

9: On a scale from 1 to 5 how difficult do you consider programming test 1 (Painters Spree)?



Very easy: 1	**0**	0%
2	**3**	42.9%
3	**3**	42.9%
4	**0**	0%
Very difficult: 5	**1**	14.3%

10: If you solved test 1 please explain what you did. If not please explain what problem you encountered?

concept van classes

Made 3 painterActors each on a different part of the width(800) canvas (1st @ 1-250, 2nd @ 250-550, 3rd @ 550-800) Drew each on a different img and drew img3 first, img2 second and img1 last so they would lay on top of eachother in a correct (visible order)

Door goed de documentatie te lezen en te luisteren naar de uitleg van het feit dat er 1 canvas actor is en een aantal actors die plaatjes moeten heen en weer sturen kon ik naar wat puzzelen wel het gewenste resultaat krijgen. Dit wil niet zeggen dat ik alles meteen snapte ik bleef wat lastigheid hebben met wat je wanneer tekent en wat je wanneer doorstuurt naar de andere thread.

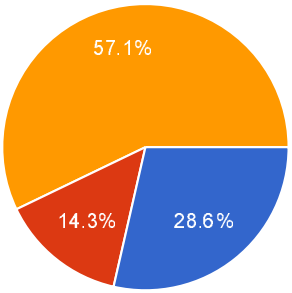
The API for the emitter/listener-pattern was not straight-forward. Other than that the painter test was not very difficult.

using mutex locks

First I setup 3 different actors to that calculate a Red green and blue value, these where painted using the canvas painter on the screen in bars, the length of the bar was the difference between the previous color value transmitted. Next I moved on to expand the the actors to draw a 100x100 object and transmit this to the canvas. The idea was to use the the delay to place them on the main screen.

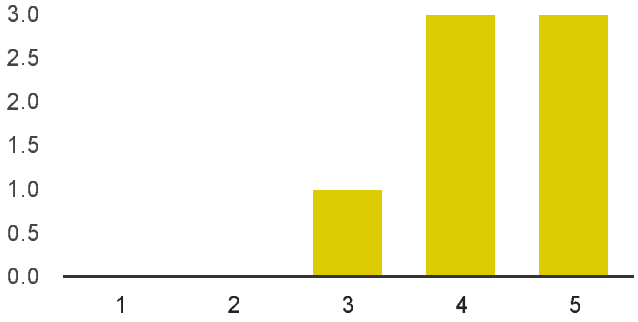
First I created a CanvasActor and one PainterActor and got that to work. I then added a second one and changed some code to accomodate for the second painter. (I did create seperate variables for the painter_image, one for painter1 (VanGogh) and one for painter2 (Rembrandt) in the CanvasActor)

11: Were you able to solve test 2 (Dining Philosophers)?



Yes	**2**	28.6%
No	**1**	14.3%
If I had more time	**4**	57.1%

12: On a scale from 1 to 5 how difficult do you consider programming test 2 (Dining Philosophers)?



Very easy: 1	**0**	0%
2	**0**	0%
3	**1**	14.3%
4	**3**	42.9%
Very difficult: 5	**3**	42.9%

13: If you solved test 2 please explain what you did. If not please explain what problem you encountered?

Again using mutex locks or a semaphore

concept van classes

I was on my way to implementing a round-robin esque chopstick passer of 3 of the five chopsticks where each philo would pass 1 chopstick after eating and the waiter would pass the remaining 2 chopsticks to the most two hungry philosophers

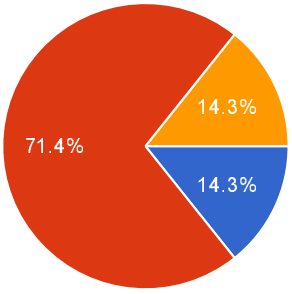
The first step was to get my head around using events (signals) to communicate between the Waiter (LeadActor) and the Philosophers. I quickly realized I needed to

use the same approach I used for Test1; set up communications between the waiter and *one* philospher and then add more philosophers. Once I got the event-based communications working it was quite easy.

In this test I only got around to setup the philosopher actors and the waiter. However when I started to think about implementing the "hand over the copsitcks" routine the time was up.

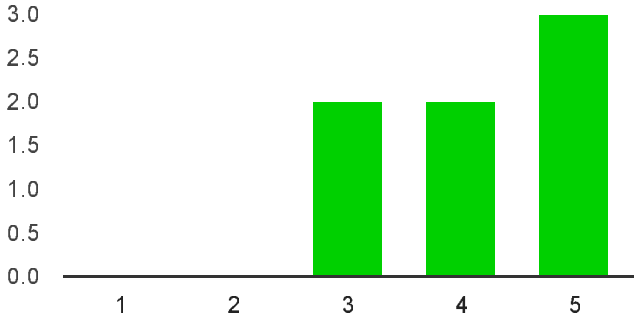
Het probleem was vooral om het gesteld probleem om te zetten in programmeer logica en blokken. Wat voor structuur heb ik nodig en als die er zijn hoe moeten ze dan met elkaar communiceren? Wat luistert naar wat en wat zijn dan de voorwaarden? Dat vond ik erg lastig. Waarom kunnen ze niet allemaal aan het denken zijn of allemaal aan het eten zijn. Hoe bepaal ik wie de chopsticks krijgt en wanneer geef ik ze terug ?

14: Were you able to solve test 3?



Yes	**1**	14.3%
No	**5**	71.4%
If I had more time	**1**	14.3%

15: On a scale from 1 to 5 how difficult do you consider programming test 3?



Very easy: 1	**0**	0%
2	**0**	0%
3	**2**	28.6%

4	**2**	28.6%
Very difficult: 5	**3**	42.9%

16: Can you explain what happens in the program of test3 and what can cause problems?

concept van classes

Met alles wat we tot dan toe geleerd hadden en de uitleg van Arnaud was de structuur met een hoofd acteur en een aantal kind acteurs duidelijk zeker ook omdat er al een voorbeeld bij zat door op dit voorbeeld door te bouwen krijg je snel in de gate wat wel en wat niet werkt en waar het mis gaat. De volgende stap is de waarom en die duurde iets langer, maar was wel logisch gebruik makende van de kennis over concurend programmeren die ik tot dan toe had geleerd. Hier had ik zeker nog wel even mijn tanden in willen zetten

Time was running out and I did not wrap my head around the popping problem yet.

no idea

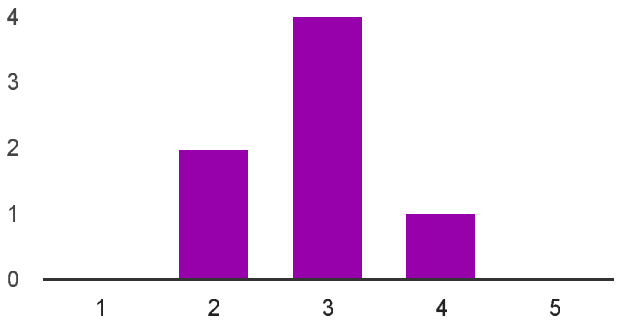
Each thread tries to pop data from the shared namespace, thus breaking the program.

The problem is when different threads use the same memory space, and these individual threads are managing and changing the information located there. It's like different cooks standing around a couldren adding, removing, emptying,sampling serving the contents without being aware of each other, but also an (maybe) algoritms in the OS or implementation of python assuming only one cook and one couldren.

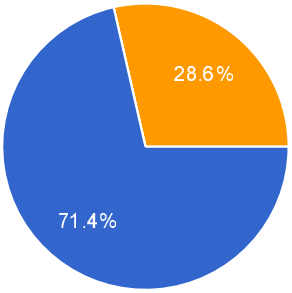
I'm not at all familiar with openCV so I needed a lot of time to understand what was happening in the first place (which function did what, etc). By the time some of the other participants had already tried one or two solutions I barely understood the code :P

Framework

17: On a scale from 1 to 5 how difficult do you find programming using the provided framework and tools?



18: Could you see yourself using this framework?



Yes	**5**	71.4%
No	**0**	0%
I'm not sure	**2**	28.6%

19: Was the provided documentation clear enough?

clear separation of concerns

Ja op zich wel soms het is duidelijk en overzichtelijk. Het vraagt naar meer. Qua classes en functionaliteit is het zeker voldoende qua overal structuur en functionaliteiten van het framework van wat je wel of wat je niet kan en wat je er waarom mee kan zou nog wel wat meer mogen zijn.

Yes. Especially the combination of a (working) example and the docs for the functions was useful. I did have some headstart because I am (somewhat) familiar with zocp

yes, but I would like to have a little more different examples

Ik miste concrete voorbeeld code.

I had severe trouble understanding the communication between threads, especially the pattern for labelling the threads.

Honestly i did not read the documentation very thouroghly, but only glanced it. Also working with a new IDE proved some bit of a hassle because the indentation of the documentation was different from the settings in the IDE and i did not know the auto indentation shortcuts.

20: What do you like best about the framework?

It basically handles the events for you, you just need to think about *what* you want to communicate and *when* you need to do that.

The ease of use and similarity with ZOCP (that I knew beforehand)

It's python, and it all seemed to just work.

Ik vind het tof dat het door bouwt op ZOCP wat ik al ken. Het is tof dat je een visueel overzicht hebt van de actors en wat ze doen. Het is handig dat je lijntjes kan tekenen. Met relatief weinig code kan je al wat maken. Het is overzichtelijk gestructureerd.

De node editor

I like that it allows me to build a performance-critical application on a cheap device like raspberry pi. I also like that you get threaded I/O for free. I look forward to exploring the new types of creative coding applications this technology facilitates.

21: What do you like least about the framework?

Waar is het framework voor? Is het om iets te maken of is het om iets te testen ? Als het is om te testen zou ik meer testen willen met een opbouw en referentie om hypotheses te staven of concepten/paradigma's te oefenen. Als het is om iets te maken wat kan ik er dan mee maken? Visueel grafisch is het nog een beetje beperkt en ik merk door de voorbeelden dat ik dan even niet zo snel inspiratie heb wat ik er nog meer mee kan maken. Ik zou wel vanuit de actor editor rechtstreeks een actor willen kunnen maken die dan ene template met file genereert waarop ik kan door bouwen.

I did not delve deep in the framework itself so i can not comment very much. However it looks like the framework, and the examples, assume that there are actors are always running and doing stuff, however in reality when i use threads its to calculate something that i do not need to be done on the main thread and end when ready

not being able to send/signal any object

I have no idea how I would need to communicate between threads without the provided framework

1.The API needs work. 2.I would prefer this framework over Processing or OpenFrameworks if it were more complete in terms of drawing and image manipulation.

It seems to be quite heavy under a light load, but sometimes that's just you not updating frequent enough (but if you do it to frequently you kill the Pi)

Abstractie niveau van Documentatie, behoefte aan in ieder geval kleine stukjes concrete syntaxis

22: How would you describe the programming using the provided framework in the usertest?

Challenging but not unfathomable

I guess it's fairly easy but it helps if you have some understanding of programming stuff that needs to communicate over a network which is similar to communicating

between threads.. I think

In mijn geval kan ik daar niets zinnigs over zeggen

Challenging, in a good way. Parallel processing is generally interesting and this framework creates a practical application I am familiar with.

The main problem was that i found it very slow to start new program's, in the philosophers example it took several second for all of the philosophers to load and connect. Also it was hard for mee to keep track of the names of the object, method of the different actors and conductors i created.

programming small sequential entities while leaving their dependency and communication to the framework

Ik zou zeggen het voelt als logisch puzzelen. Een soort DenkSport puzzelboeken maar dan voor programmeren. Als ik gewoon wat wil puzzelen en programmeren dan zou ik dit framework pakken. Dus dat leidt tot de vraag wat is de scope van het framework? Mag duidelijker gesteld worden wat mij betreft. Wat wil je ermee? Wat kan je ermee? Waar wil je ermee naar toe? Waarom zou ik als programmeur het willen gebruiken/ermee spelen ?

Thank you

If you have any suggestions or remarks please leave them below

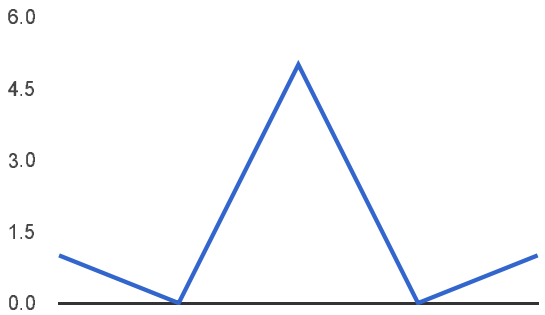
Ik vond het super leuk om mee te doen. Ik ben altijd bereid om er over door te kletsen met een biertje erbij en ben erg nieuwsgierig naar de rest van het onderzoek. Vandaag was helder duidelijk en goed gestructureerd. Het is echt ene framework met alles erop en eraan. Niet gek om dat zo even uit de grond te stampen...

Meer voorbeelden in documentatie.

keep me posted!

None, thank you for the lunch

Number of daily responses



10.3 Appendix C: User Test Documentation

For the user test the following documentation was written which the subjects could consult. This was written as a guide as well as typical programming API documentation. As the introductory workshop only covered the framework as a whole some finer details were noted in the documentation.

Creative Concurrency Documentation

Release 0.1

Arnaud Loonstra

June 27, 2015

Contents

This is the documentation of a simple toolkit provided to research concurrent programming for Creative Coders. The toolkit is build upon the ZOCP framework.

You can find the API documentation as well as simple examples of the toolkit's usage.

CHAPTER 1

Installation

Note: You're machine should come preinstalled with the `sphof` module.

1.1 Operating System installation

You should install the latest Python version on your machine

For the Pillow module you'll need to make sure you have the Tcl/Tk libraries and includes installed

```
$ sudo apt-get install tcl-dev tcl tk tk-dev python3-tk
```

1.2 Required Python modules

The `sphof` module requires the following modules:

- Pillow
- Pyre
- pyZOCP

You can install these using 'pip':

```
$ pip install Pillow
$ pip install pyzmq
$ pip install https://github.com/zeromq/pyre/archive/master.zip
$ pip install https://github.com/z25/pyZOCP/archive/master.zip
```

Note: On some operating systems 'pip' is named 'pip3' or 'pip-3.2'

CHAPTER 2

Guide

In this test we will create programs which enable the use of multiple processors of a computer. You will be provided with a framework and some tools to create programs for the assignments.

This website provides the descriptions of the assignments as well as a reference for the framework and tools.

2.1 Introduction

In the framework we call a small program an 'Actor'. The `sphof` framework provides different 'Actor' classes. These classes have a setup(), update() and draw() method similar to OpenFrameworks and Processing.

Additionally to these methods there are methods to enable communication between the Actors. This communication is done using signals which you are going to use during the assignments.

In the Actor classes you can register named variables to be used for communication with other Actors. I.e. to register an integer:

```
1 class MyFirstActor(Actor):
2
3     def setup(self):
4         self.register_int("MyFirstInt", 0, "re")
5
6     def update(self):
7         self.emit_signal("MyFirstInt", self.get_value("MyFirstInt")+1)
```

It's important to understand that once an Actor has a variable registered every other Actor can access this value. However before acquiring the value of a variable the Actor interested in the variable first needs to subscribe to it. This can be accomplished by using the signal_subscribe method. I.e:

```
1 class MySecondActor(Actor):
2
3     def on_peer_enter(self, peer, name, headers, *args, **kwargs):
4         self.signal_subscribe(self.uuid(),  None, peer, "MyFirstInt")
5
6     def on_peer_signaled(self, peer_id, name, signal):
7         print(name, signal)
```

By subscribing to the MyFirstInt variable of the MyFirstActor the MyFirstActor will send the value of the variable through a signal. Of course you first need to be aware of the MyFirstActor, hence the usage of the `on_peer_enter` method. Remember as we are running Actors on multiple processors you will never know if your program started first or if the other was first. Therefore the `on_peer_enter` method will tell you.

It might also be easier to directly link variables of Actors. You can do this by registering a variable and then subscribing this variable to another Actor's variable. In the MySecondActor example we can do this as follows:

```
1  class MySecondActor(Actor):
2
3      def setup(self):
4          self.register_int("MySecondInt", 0, "rs")
5
6      def on_enter_peer(self, peer, name, headers, *args, **kwargs):
7          self.signal_subscribe(self.uuid(),  "MySecondInt", peer, "MyFirstInt")
8
9      def update(self):
10         print(self.get_value('MySecondInt'))
```

Note: Notice the difference in the last parameter of the register_int method of both classes. In the MyFirstActor class it is 're' and in the MySecondActor it is 'rs'. 'r' Means the variable is **r**eadable. 's' Means the variable is a 'signal **s**ensor'. This implies it can receive signals. 'e' Means the variable is a 'signal **e**mitter'. It means the variable can send signals. Read more about this in the `ZOCP reference`.

2.2 Starting Actors

We now know how to program Actors and let them communicate with each other. However, we still need to start them. It's important to know that any regular program always has one 'main' thread. Only from the 'main' thread you can start other threads in order to utilize multiple processors. For the 'main' thread we use the `LeadActor` class which provides us methods for starting more `Actor` instances. Remember you can only have **one** LeadActor in your program!

For example a simple LeadActor looks like this:

```
1  from sphof import LeadActor
2
3  class MyLeadActor(Actor):
4
5      def setup(self):
6          self.register_int("MyLeadInt", 0, "rs")
7
8      def update(self):
9          print(self.get_value("MyLeadInt"))
10
11 app = MyLeadActor('MyLeadActor')
12 app.run()
```

Save this text as 'myapp.py'. You can then run this program as follows:

```
$ python3 myapp.py
```

It will print repeating lines of '0'. You can stop the program by sending a KeyboardInterrupt. Just press the CTRL-C keyboard combination.

Note: You can also directly execute from Geany however it is important to understand this is exactly the same as running from a terminal.

Also notice line 9 where we instantiate the MyLeadActor class and providing "MyLeadActor" as an argument. Every Actor needs a name. You can provide the name as a first argument when you instantiate the Actor instance. If you don't provide a name a random name will be made up!

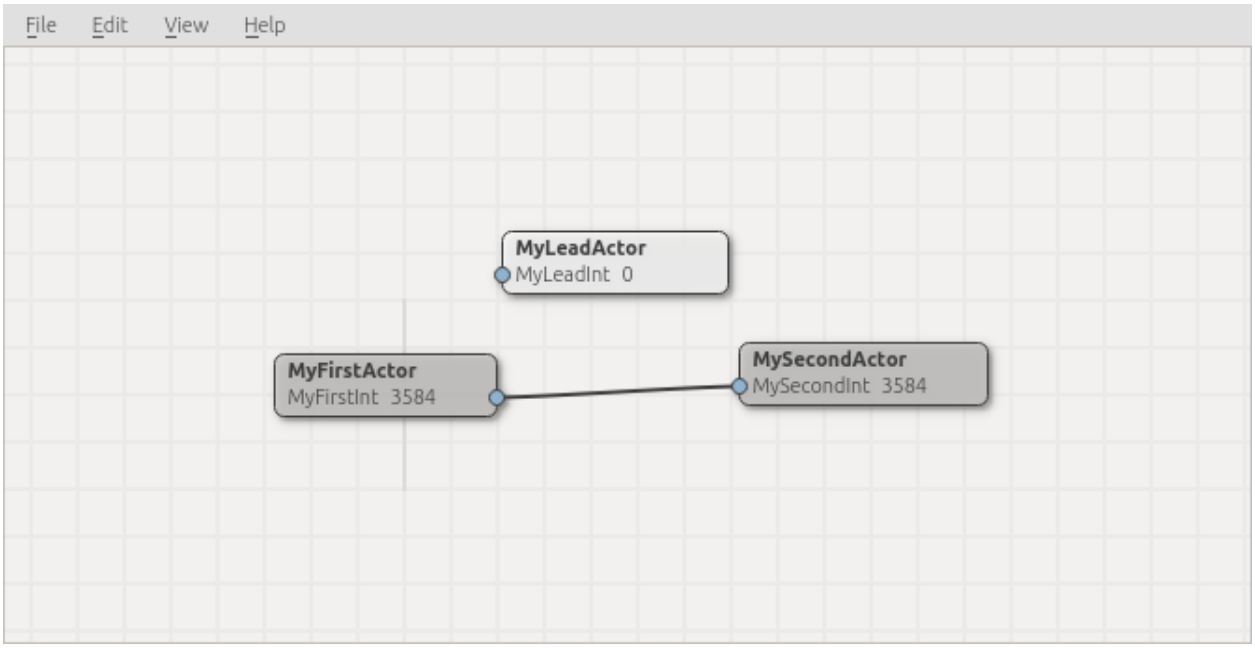
Now if we would want to run the MyFirstActor and MySecondActor we can use the MyLeadActor class as follows:

```
1  from sphof import *
2
3  class MyFirstActor(Actor):
4
5      def setup(self):
6          self.register_int("MyFirstInt", 0, "re")
7
8      def update(self):
9          self.emit_signal("MyFirstInt", self.get_value("MyFirstInt")+1)
10
11
12 class MySecondActor(Actor):
13
14     def setup(self):
15         self.register_int("MySecondInt", 0, "rs")
16
17     def on_peer_enter(self, peer, peer_name, *args, **kwargs):
18         if peer_name == "MyFirstActor":
19             self.signal_subscribe(self.uuid(),  "MySecondInt", peer, "MyFirstInt")
20
21     def update(self):
22         print(self.get_value('MySecondInt'))
23
24
25 class MyLeadActor(LeadActor):
26
27     def setup(self):
28         self.add_actor(MyFirstActor('MyFirstActor'))
29         self.add_actor(MySecondActor('MySecondActor'))
30         self.register_int("MyLeadInt", 0, "rs")
31
32     def update(self):
33         return
34         print(self.get_value("MyLeadInt"))
35
36
37 app = MyLeadActor('MyLeadActor')
38 app.run()
```

Note: Line 18 is different from the original MySecondActor. This is because we now run 3 Actors and we only want to subscribe the MyFirstActor to the MySecondActor. Therefore we need to test which Actor we are dealing with in the `on_peer_enter` method.

2.3 Visualizing and editing Actors

Ok, we now know how to program Actors and how to run them. Now imagine a whole lot of them. To be able to oversee how all Actors relate to each other we have a visualization tool. On your system you can find the ActorEditor.



Just start the ActorEditor and it will display any Actors you have running. You can make subscriptions between Actors bij dragging a line between Actor's emitters and sensors. Emitters are always on the right side of an Actor representation. Sensors are on the left.

Now make sure you run the LeadActor example we just discussed. The ActorEditor will display them like in the screenshot. Try to subscribe the MyFirstActor's MyFirstInt to the LeadActor's LeadInt. You do this by dragging a line from the emitter to the sensor. This manual action is equal to the code:

```
self.signal_subscribe(<LeadActor>.uuid(),  "MyLeadInt", <MyFirstActor>.uuid(), "MyFirstInt")
```

Note: Of course you need to replace <LeadActor> and <MyFirstActor> with the right names in your code

CHAPTER 3

Test 1: Painters Spree

Imagine you have created a simple application that draws something on the screen. Your processor is not fast enough to draw 60 frames per second.

In this first test we will create a program which handles multiple painters. This is often a problem in concurrent programs as OpenGL and most graphic libraries can only run in the main thread. Therefore it is impossible to let multiple Actors draw on the display. We will need to workaround this limitation.

You need to use the *Canvas Actor classes* for these have simple methods for drawing. First start by creating a simple painter using the `sphof.LoneActor` class:

```
from sphof import LonePainterActor
from random import randint

class SinglePainter(LonePainterActor):

    def setup(self):
        self.set_width(800)
        self.set_height(600)

    def draw(self):
        start = (
            randint(0, self.get_width()),    # x coordinate
            randint(0, self.get_height())    # y coordinate
            )
        end = (
            randint(0, self.get_width()),    # x coordinate
            randint(0, self.get_height())    # y coordinate
            )
        color = (
            randint(70,110),                 # red color
            randint(160,210),                # green color
            randint(70,210)                  # blue color
            )
        self.line([start, end], color, 20)

painter = SinglePainter("SinglePainter")
painter.run()
```

This runs on a single processor. Now if we would want to have multiple painters using multiple processors we need to create an Actor for displaying and other Actors for creating the drawings. As you read in the *guide* you can use a `LeadActor` to start other Actors. You can now understand that this `LeadActor` also needs to display the drawings as it will be the only Actor with access to the display of the computer!

3.1 PainterActor and CanvasActor Class

The `PainterActor` class provides a `send_img` method for signalling a new image. The `PainterActor` class also automatically registers the 'imgID' variable which is a reference to the image. Therefore you can simply call `send_img` to send the image. However there is one rule of thumb: Once you send the image you do not own it anymore!

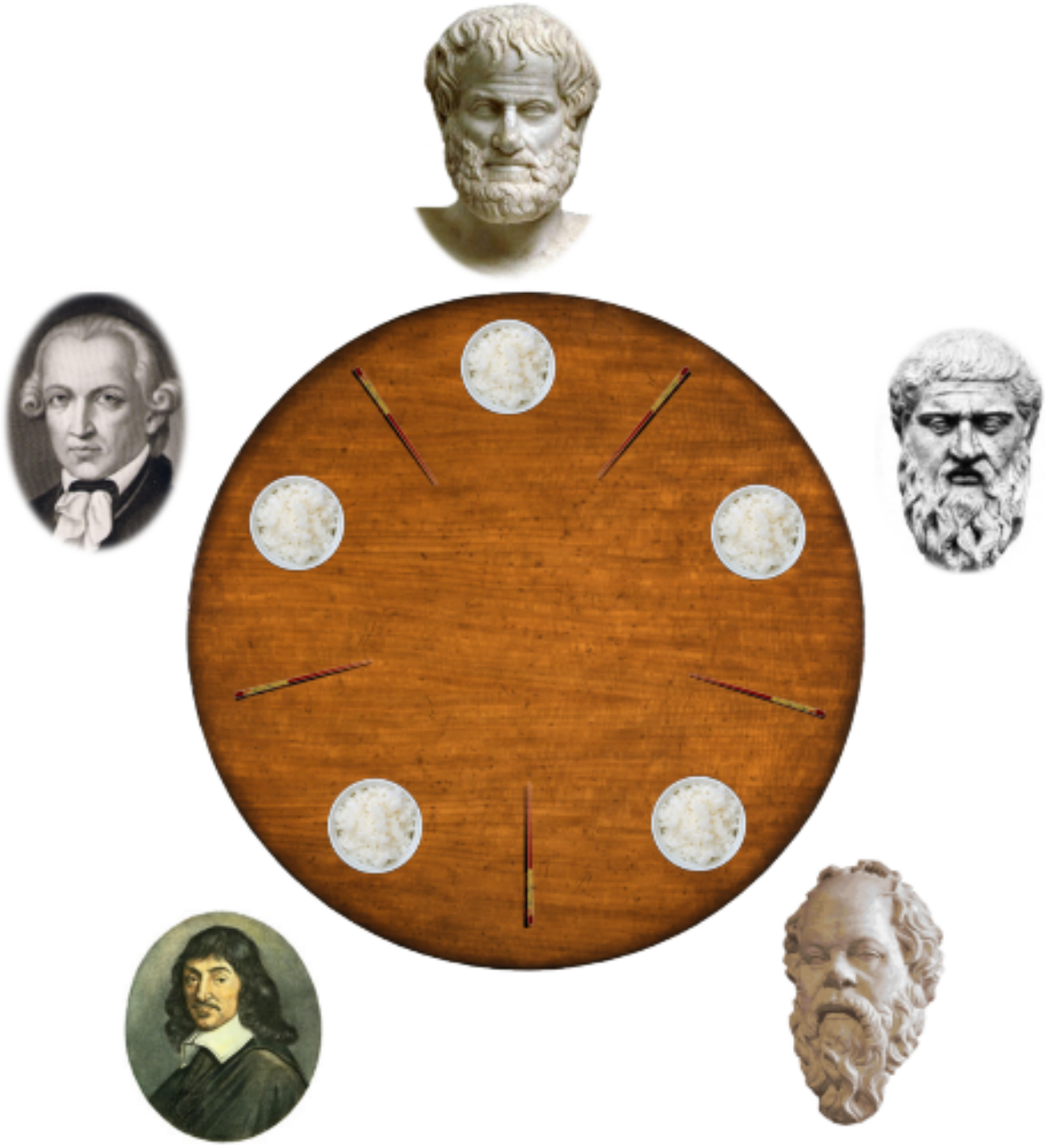
The `CanvasActor` class provides a `get_img_from_id` method. You can pass the imgID value and it will return the image. You can then use `draw_img` to display the image.

Why these methods? You have to understand that you cannot just pass images around like that. An image occupies a large amount of memory and copying them takes a large amount of time. Therefore the sending happens by passing a reference instead of the full image. In languages like C or C++ you'd call this a pointer. This is a bit difficult in a language like Python because if we would send the image it will be garbage collected after being send. Anyway, these are just convenience methods to prevent you from running into trouble and keeping your machine performant.

CHAPTER 4

Test 2: Dining Philosophers

In the second test we will search for a solution of a typical computer science problem. Five philosophers sit at a round table with bowls of rice. Chopsticks are placed between each pair of adjacent philosophers.



Each philosopher must alternately think and eat. However, a philosopher can only eat rice when he has both left and right chopstick. Each chopstick can be held by only one philosopher and so a philosopher can use the chopstick only if it is not being used by another philosopher. After he finishes eating, he needs to put down both chopsticks so they become available to others. A philosopher can take the chopstick on his right or the one on his left as they become available, but cannot start eating before getting both of them.

There is an infinite amount of rice in the bowls.

You need to design a program which makes sure all philosophers can think and eat. There are many solutions to this problem but you are adviced to use a waiter which serves the table.

In the framework a `PhilosopherActor` class is provided. This actor has the methods `think` and `eat`. If a philosopher is in the thinking state the `think` method needs to be called. If the philosopher is in the eating state the `eat` method needs to be called. A single philosopher implementation is given below:

```
import time
from sphof import LonePhilosopherActor

class SinglePhilosopher(LonePhilosopherActor):

    def setup(self):
        self.state_hungry = True
        self.switch_at = time.time() + 5      # switch state every 5s
        self.enlightenment = None
        self.topics = []                      # food for thought

    def update(self):
        if time.time() > self.switch_at or not(len(self.topics)):
            # it's time to switch state
            self.switch_at = time.time() + 5    # set next state switch timestamp
            self.state_hungry = not(self.state_hungry)
            if self.state_hungry:
                print("Jay food! Eating....", len(self.topics))
            else:
                print("Hmmmmm... let me think...", len(self.topics))

        if not self.state_hungry:
            enlightenment = self.think()
            if enlightenment:
                print("Eureka:", enlightenment, len(self.topics))
        else:
            self.eat()


if __name__ == '__main__':
    test = SinglePhilosopher("Descartes")
    test.run()
```

You can use this implementation for your multiple philosophers implementation.

CHAPTER 5

Test 3: It works... most of the time

In the third test the source of classes is given. You are asked to create a program similar to the given screenshot below:

As you can see the application captures from a camera, displays this in a window and also displays three thumbnails of the same video in the corner.

Sample classes are given in the example below:

```
import sphof
from sphof import LeadActor, Actor
import cv2
import numpy as np
```

```
class OpenCVActor(Actor):

    def setup(self):
        self.register_int("img_in", 0, "rs")
        self.register_int("img_out", 0, "re")

    def send_img(self, img, ID):
        """
        Sends the image as a signal to any subscribers using the 'imgID'
        emitter. The canvas is reset after the image is sent!
        """
        imgID = id(img)
        assert(imgID not in sphof.shared_ns)
        sphof.shared_ns[imgID] = img
        self.emit_signal(ID, imgID)

    def get_img_from_id(self, imgID):
        """
        Get the image from the given imgID
        """
        return sphof.shared_ns.pop(imgID)

    def resize(self, img, width, height):
        return cv2.resize(img, (width, height))

    def invert(self, img):
        return 255-img

    def to_hsv(self, img):
        return cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    def blur(self, img):
        kernel = np.ones((5,5),np.float32)/25
        return cv2.filter2D(img,-1,kernel)

    def on_peer_signaled(self, peer, name, data):
        imgID = self.get_value("img_in")
        img = self.get_img_from_id(imgID)
        img_s = self.resize(img, 120, 90)
        self.send_img(img_s, "img_out")


class CVCapLeadActor(LeadActor):

    def setup(self):
        self.add_actor(OpenCVActor("CVActor"))
        self.video_capture = cv2.VideoCapture(0)
        #self.video_capture.set(cv2.CAP_PROP_FRAME_WIDTH, 320)
        #self.video_capture.set(cv2.CAP_PROP_FRAME_HEIGHT, 240)
        #self.video_capture.set(cv2.CAP_PROP_FPS, 15)
        self.frame = None
        self.thumb = None
        self.register_int("imgID_out", 0, "re")
        self.register_int("thumb_in", 0, "rs")
        cv2.startWindowThread()
        cv2.namedWindow('Video')
        self.cap_success = False
```

```
    def update(self):
        self.cap_success, self.frame = self.video_capture.read()
        if self.cap_success:
            self.send_img(self.frame, "imgID_out")

    def draw(self):
        if self.thumb != None:
            self.frame[0:90, 0:120] = self.thumb
            cv2.imshow('Video', self.frame)

    def send_img(self, img, ID):
        """
        Sends the image as a signal to any subscribers using the 'imgID'
        emitter. The canvas is reset after the image is sent!
        """
        imgID = id(img)
        sphof.shared_ns[imgID] = img
        self.emit_signal(ID, imgID)

    def on_peer_enter(self, peer, name, headers):
        if name == "CVActor":
            self.signal_subscribe(self.uuid(), "thumb_in", peer, "img_out")
            self.signal_subscribe(peer, "img_in", self.uuid(), "imgID_out")

    def on_peer_signaled(self, peer, name, date):
        if name == "CVActor":
            self.thumb = sphof.shared_ns.pop(self.get_value('thumb_in'))

    def stop(self):
        self.video_capture.release()
        cv2.destroyAllWindows()
        super(CVCapLeadActor, self).stop()


if __name__ == "__main__":
    lead_actor = CVCapLeadActor("CVCaptureActor")
    lead_actor.run()
```

Now try to create more thumbnails of the video.

Note: What is the difficulty in this program? What are best approaches? It is normal to end up in discussion in this assignment.

CHAPTER 6

Survey

Once you are done with all three assignments please fill in the survey:

Survey at Google Docs

CHAPTER 7

sphof module reference

7.1 Actor classes

7.1.1 Frequently used methods

sphof.Actor(*args, **kwargs)	An Actor class runs inside its own thread.
sphof.Actor.setup()	Called a startup.
sphof.Actor.update()	Called every loop
sphof.Actor.draw()	Called after update
sphof.LeadActor.add_actor(actor)	Add an Actor and run its threaded loop
sphof.LeadActor.run()	Run the actor's application loop
sphof.LeadActor.stop()	Stop this LeadActor.

7.1.2 Actor class

class **Actor**(**args*, ***kwargs*)

An Actor class runs inside its own thread. It's usually started by a LeadActor!

Parameters **name** (*str*) – Name of the node, if not given a random name will be created

By default the Actor loop runs at 60 iterations per second. This means your update and draw method is called every 1/60th second.

- Use the `Actor.setup()` method to setup the class
- Use the `Actor.update()` method to update anything you have setup
- Use the `Actor.draw()` method to visualize

Warning: It is important to understand that an actor runs in a thread. Usually a thread is started by a 'main' thread. A `sphof.LeadActor` provides methods for starting and stopping Actors as the LeadActor runs in the main thread. An Actor has limitations. For example you cannot visualize directly from an Actor. To visualize what an actor draws you'll need to handover the image to a LeadActor.

setup()

Called a startup.

Add variables you want to use througout the actor here. I.e.:

```
self.count = 0
```

and in the update() method:

```
self.count += 1
```

update()
Called every loop

draw()
Called after update

7.1.3 LeadActor class

class LeadActor(**args*, ***kwargs*)
Bases: sphof.actors.Actor

A LeadActor class runs in the main thread. It inherits all methods from the Actor class but has some additional methods to start Actors

Parameters **name** (*str*) – Name of the node, if not given a random name will be created

By default the LeadActor loop runs at 60 iterations per second. This means your update and draw method is called every 1/60th second.

- Use the `Actor.setup()` method to setup the class
- Use `Actor.update()` method to update anything you have setup
- Use `Actor.draw()` method to visualise

stop()
Stop this LeadActor. Before stopping all Actors started from this LeadActor are stopped first

add_actor(*actor*)
Add an Actor and run its threaded loop

Parameters **actor** (*Actor*) – An Actor to start in its own thread

draw()
Called after update

remove_actor(*actor*)
Remove and stop an Actor

Parameters **actor** (*Actor*) – An Actor to remove and stop

run()
Run the actor's application loop

setup()
Called a startup.

Add variables you want to use througout the actor here. I.e.:

```
self.count = 0
```

and in the update() method:

```
self.count += 1
```

update()
Called every loop

7.1.4 LoneActor class

class LoneActor (*name*, **args*, ***kwargs*)

The LoneActor class runs an application loop.

Parameters **name** (*str*) – Name of the node, if not given a random name will be created

By default the LoneActor loop runs at 60 iterations per second. This means your update and draw method is called every 1/60th second.

- Use the `LoneActor.setup()` method to setup the class
- Use `LoneActor.update()` method to update anything you have setup
- Use `LoneActor.draw()` method to visualise

setup()

Called a startup.

Add variables you want to use througout the actor here. I.e.:

```
self.count = 0
```

and in the update() method:

```
self.count += 1
```

update()

Called every loop

draw()

Called after update

7.2 Canvas Actor classes

The Canvas Actor classes provide classes to create drawings/images and to display them.

7.2.1 PainterActor class

class PainterActor (**args*, ***kwargs*)

Bases: `sphof.canvas_actors.Painter`, `sphof.actors.Actor`

The PainterActor class is an `Actor` with all the `Painter` class's methods and providing methods to handover the image to a LeadActor.

example:

```
from sphof import PainterActor
from random import randint

class MyPainter(PainterActor):

    def setup(self):
        self.count = 0                  # initialize counter

    def update(self):
        self.count += 1                 # increment counter
        if self.count == self.get_width():
            self.count = 0              # reset counter
```

```
            self.send_img()                 # emit the imgID so a
                                            # LeadActor could
                                            # display it
    def draw(self):
        start = (self.count, 0)             # start position
        end = ( self.count,
                self.get_height())          # end position
        color = (
                randint(7,210),             # red
                randint(16,210),            # green
                randint(70,210)             # blue
                )
        self.line((start, end), color, 2)# draw line
```

To display the PainterActor's drawing you need to send the image to `CanvasActor` which can display the image on screen. In order to send an image use the `send_img()` method.

The `send_img()` method emits a 'imgID' signal containing a pointer to the image of this Actor. It calls reset() so the actor can paint on a new canvas.

This class has many methods inherited from the `sphof.Painter` class, ie:

- `line()`
- `rectangle()`
- `ellipse()`
- `arc()`

Each class's extra methods are documented below.

send_img()
Sends the image as a signal to any subscribers using the 'imgID' emitter. The canvas is reset after the image is sent!

7.2.2 CanvasActor class

class CanvasActor(**args*, ***kwargs*)
Bases: `sphof.canvas_actors.Painter`, `sphof.actors.LeadActor`

The CanvasActor class implements methods for drawing on a canvas (screen) similar to the `PainterActor`

To display drawings of PainterActors you need to receive the image of a PainterActor by subscribing to the 'imgID' signal emitter of the PainterActor.

example:

```
from sphof import CanvasActor

class MyCanvas(CanvasActor):

    def setup(self):
        self.painter_img = None
        self.register_int("PaintingID", 0, "rs") # create a sensor for image ids
        # ... setup the painter here

    def on_peer_enter(self, peer, name, headers):
        if name == "PainterName":       # PainterName is the name of your PainterActor
            self.signal_subscribe(self.uuid(), "PaintingID", peer, "imgID")
```

```
        def on_peer_signaled(self, peer, name, data):
            if name == "PainterName":
                self.painter_img = self.get_img_from_id(self.get_value("PaintingID"))

        def draw(self):
            if self.painter_img:
                self.draw_img(self.painter_img)
    la = MyCanvas()
    la.run()
```

get_img_from_id(*imgID*)
Get the image from the given imgID

draw_img(*img*, *x=0*, *y=0*)
Draw the image at position x,y

7.2.3 LonePainterActor class

class LonePainterActor(**args*, ***kwargs*)
Bases: sphof.canvas_actors.Painter, sphof.actors.LoneActor

7.2.4 Painter class

class Painter(**args*, ***kwargs*)
The Painter class provides simple methods for drawing, ie:

- line()
- rectangle()
- ellipse()
- arc()

The default width and height are 200 by 600 pixels.

Each class's method is documented below

reset()
Clears the image to the background color

get_width()
Returns the width of the canvas

set_width(*width*)
Set the width of the canvas, it will reset your image! :param width: Width of the canvas in pixels

get_height()
Returns the height of the canvas

set_height(*height*)
Set the height of the canvas, it will reset your image! :param width: Width of the canvas in pixels

arc(**args*, ***kwargs*)
Draws an arc (a portion of a circle outline) between the start and end angles, inside the given bounding box.

Parameters

- **xy** – Four points to define the bounding box. Sequence of `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`.
- **start** – Starting angle, in degrees. Angles are measured from 3 o'clock, increasing clockwise.
- **end** – Ending angle, in degrees.
- **fill** – Color to use for the arc.

bitmap(**args*, ***kwargs*)

Draws a bitmap (mask) at the given position, using the current fill color for the non-zero portions. The bitmap should be a valid transparency mask (mode "1") or matte (mode "L" or "RGBA").

This is equivalent to doing `image.paste(xy, color, bitmap)`.

To paste pixel data into an image, use the `paste()` method on the image itself.

chord(**args*, ***kwargs*)

Same as `arc()`, but connects the end points with a straight line.

Parameters

- **xy** – Four points to define the bounding box. Sequence of `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

ellipse(**args*, ***kwargs*)

Draws an ellipse inside the given bounding box.

Parameters

- **xy** – Four points to define the bounding box. Sequence of either `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

line(**args*, ***kwargs*)

Draws a line between the coordinates in the **xy** list.

Parameters

- **xy** – Sequence of either 2-tuples like `[(x, y), (x, y), ...]` or numeric values like `[x, y, x, y, ...]`.
- **fill** – Color to use for the line.
- **width** – The line width, in pixels. Note that line joins are not handled well, so wide polylines will not look good.

pieslice(**args*, ***kwargs*)

Same as arc, but also draws straight lines between the end points and the center of the bounding box.

Parameters

- **xy** – Four points to define the bounding box. Sequence of `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`.
- **start** – Starting angle, in degrees. Angles are measured from 3 o'clock, increasing clockwise.
- **end** – Ending angle, in degrees.

- **fill** – Color to use for the fill.
- **outline** – Color to use for the outline.

point (**args*, ***kwargs*)
Draws points (individual pixels) at the given coordinates.

Parameters

- **xy** – Sequence of either 2-tuples like `[(x, y), (x, y), ...]` or numeric values like `[x, y, x, y, ...]`.
- **fill** – Color to use for the point.

polygon (**args*, ***kwargs*)
Draws a polygon.

The polygon outline consists of straight lines between the given coordinates, plus a straight line between the last and the first coordinate.

Parameters

- **xy** – Sequence of either 2-tuples like `[(x, y), (x, y), ...]` or numeric values like `[x, y, x, y, ...]`.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

rectangle (**args*, ***kwargs*)

Draws a rectangle.

Parameters

- **xy** – Four points to define the bounding box. Sequence of either `[(x0, y0), (x1, y1)]` or `[x0, y0, x1, y1]`. The second point is just outside the drawn rectangle.
- **outline** – Color to use for the outline.
- **fill** – Color to use for the fill.

text (*xy*, *text*, *fill*)
Draws the string at the given position.

Parameters

- **xy** – Top left corner of the text.
- **text** – Text to be drawn.
- **fill** – Color to use for the text.

textsize (*text*)
Return the size of the given string, in pixels.

Parameters **text** – Text to be measured.

7.3 Philosopher Actor classes

The Philosopher Actor classes provide classes with methods for the Dining Philosophers Problem from test 2.

7.3.1 PhilosopherActor class

class `PhilosopherActor`(**args*, ***kwargs*)
Bases: `sphof.philosopher_actors.Philosopher`, `sphof.actors.Actor`

`draw`()
Called after update

`eat`()
The eat method makes the philosopher eat. This fills its list of topics for thinking (food for thought)

`emit_signal`(*emitter*, *value*)
Update the value of the emitter and signal all subscribed receivers

Parameters

- **emitter** (*str*) – name of the emitting variable
- **value** – the new value

`on_peer_enter`(*peer*, *name*, **args*, ***kwargs*)
This method is called when a new peer is discovered

Parameters

- **peer** (*uuid*) – the id of the new peer
- **name** (*str*) – the name of the new peer

`register_string`(*name*, *value*, *access='r'*)
Register a string variable

Parameters

- **name** (*str*) – the name of the variable as how nodes can refer to it
- **value** (*str*) – the variable value
- **access** (*str*) – set the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor

`setup`()
Called a startup.

Add variables you want to use througout the actor here. I.e.:

```
self.count = 0
```

and in the update() method:

```
self.count += 1
```

`signal_subscribe`(*recv_peer*, *receiver*, *emit_peer*, *emitter*)
Subscribe a receiver to an emitter

Parameters

- **recv_peer** (*uuid*) – the id of the receiving peer
- **receiver** (*str*) – the name of the receiving variable. If None, no capability on the receiving peer is updated, but a on_peer_signal event is still fired.
- **emit_peer** (*uuid*) – the id of the emitting peer
- **emitter** (*str*) – the name the emitter. If None, all capabilities will emit to the receiver

Note: A third node can instruct two nodes to subscribe to one another by specifying the ids of the peers. The subscription request is then sent to the emitter node which in turn forwards the subscribtion request to the receiver node.

think()
The think methods makes the philospher actor think about something and determine the quality of it. If the quality is good it will return the thought. Otherwise it returns None

If the philospher is out of topics it will say so. He then needs to eat.

update()
Called every loop

7.3.2 LonePhilosopherActor class

class LonePhilosopherActor(**args*, ***kwargs*)
Bases: `sphof.philosopher_actors.Philosopher`, `sphof.actors.LoneActor`

draw()
Called after update

eat()
The eat method makes the philosopher eat. This fills its list of topics for thinking (food for thought)

setup()
Called a startup.

Add variables you want to use througout the actor here. I.e.:

```
self.count = 0
```

and in the update() method:

```
self.count += 1
```

think()
The think methods makes the philospher actor think about something and determine the quality of it. If the quality is good it will return the thought. Otherwise it returns None

If the philospher is out of topics it will say so. He then needs to eat.

update()
Called every loop

7.4 ZOCP classes & methods

7.4.1 Frequently used methods

`zocp.ZOCP.register_int`(name, value[, ...])	Register an integer variable
`zocp.ZOCP.register_float`(name, value[, ...])	Register a float variable
`zocp.ZOCP.register_string`(name, value[, access])	Register a string variable
`zocp.ZOCP.get_value`(name)	Retrieve the current value of a named parameter in the capability tree
`zocp.ZOCP.signal_subscribe`(recv_peer, ...)	Subscribe a receiver to an emitter
`zocp.ZOCP.emit_signal`(emitter, value)	Update the value of the emitter and signal all subscribed receivers

Continued on next page

Table 7.2 – continued from previous page

zocp.ZOCP.on_peer_enter(peer, name, *args, ...)	This method is called when a new peer is discovered
zocp.ZOCP.on_peer_subscribed(peer, name, ...)	Called when a peer subscribes to an emitter on this node.
zocp.ZOCP.on_peer_signaled(peer, name, data, ...)	Called when a peer signals that some of its data is modified.

7.4.2 ZOCP class

class ZOCP (**args*, ***kwargs*)

The ZOCP class provides all methods for ZOCP nodes

Parameters **name** (*str*) – Name of the node, if not given a random name will be created

set_capability (*cap*)

Set node's capability, overwites previous :param dict cap: The dictionary replacing the previous capabilities

get_capability ()

Return node's capabilities :return: The capability dictionary

set_node_location (*location=[0, 0, 0]*)

Set node's location, overwites previous

set_node_orientation (*orientation=[0, 0, 0]*)

Set node's name, overwites previous

set_node_scale (*scale=[0, 0, 0]*)

Set node's name, overwites previous

set_node_matrix (*matrix=[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]*)

Set node's matrix, overwites previous

set_object (*name=None*, *type='Unknown'*)

Create a new object on this nodes capability

register_int (*name*, *value*, *access='r'*, *min=None*, *max=None*, *step=None*)

Register an integer variable

Parameters

- **name** (*str*) – the name of the variable as how nodes can refer to it
- **value** (*int*) – the variable value
- **access** (*str*) – the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor
- **min** (*int*) – minimal value
- **max** (*int*) – maximal value
- **step** (*int*) – step value for increments and decrements

register_float (*name*, *value*, *access='r'*, *min=None*, *max=None*, *step=None*)

Register a float variable

Parameters

- **name** (*str*) – the name of the variable as how nodes can refer to it
- **value** (*float*) – the variable value
- **access** (*str*) – the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor

- **min** (*float*) – minimal value
- **max** (*float*) – maximal value
- **step** (*float*) – step value for increments and decrements

register_percent(*name*, *value*, *access='r'*, *min=None*, *max=None*, *step=None*)
Register a percentage variable

Parameters

- **name** (*str*) – the name of the variable as how nodes can refer to it
- **value** (*float*) – the variable value
- **access** (*str*) – the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor
- **min** (*float*) – minimal value
- **max** (*float*) – maximal value
- **step** (*float*) – step value for increments and decrements

register_bool(*name*, *value*, *access='r'*)
Register an integer variable

Arguments are: :param str name: the name of the variable as how nodes can refer to it :param bool value: the variable value :param str access: the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor

register_string(*name*, *value*, *access='r'*)
Register a string variable

Parameters

- **name** (*str*) – the name of the variable as how nodes can refer to it
- **value** (*str*) – the variable value
- **access** (*str*) – set the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor

register_vec2f(*name*, *value*, *access='r'*, *min=None*, *max=None*, *step=None*)
Register a 2 dimensional vector variable

Parameters

- **name** (*str*) – the name of the variable as how nodes can refer to it
- **value** (*tuple*) – the variable value
- **access** (*str*) – the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor
- **min** (*tuple*) – minimal value
- **max** (*tuple*) – maximal value
- **step** (*tuple*) – step value for increments and decrements

register_vec3f(*name*, *value*, *access='r'*, *min=None*, *max=None*, *step=None*)
Register a three dimensional vector variable

Parameters

- **name** (*str*) – the name of the variable as how nodes can refer to it

- **value** (*tuple*) – the variable value
- **access** (*str*) – the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor
- **min** (*tuple*) – minimal value
- **max** (*tuple*) – maximal value
- **step** (*tuple*) – step value for increments and decrements

register_vec4f (*name*, *value*, *access='r'*, *min=None*, *max=None*, *step=None*)
Register a four dimensional vector variable

Parameters

- **name** (*str*) – the name of the variable as how nodes can refer to it
- **value** (*tuple*) – the variable value
- **access** (*str*) – the access state of the variable. 'r'=readable, 'w'=writeable, 'e'=signal emitter, 's'=signal sensor
- **min** (*tuple*) – minimal value
- **max** (*tuple*) – maximal value
- **step** (*tuple*) – step value for increments and decrements

get_value (*name*)
Retrieve the current value of a named parameter in the capability tree

Parameters **name** (*str*) – the name of the variable as how nodes refer to it

Returns the value of the named variable

peer_get_capability (*peer*)
Get the capabilities of peer

Convenience method since it's the same a calling GET on a peer with no data

peer_get (*peer*, *keys*)
Get items from peer

peer_set (*peer*, *data*)
Set items on peer

peer_call (*peer*, *method*, **args*)
Call method on peer

signal_subscribe (*recv_peer*, *receiver*, *emit_peer*, *emitter*)
Subscribe a receiver to an emitter

Parameters

- **recv_peer** (*uuid*) – the id of the receiving peer
- **receiver** (*str*) – the name of the receiving variable. If None, no capability on the receiving peer is updated, but a on_peer_signal event is still fired.
- **emit_peer** (*uuid*) – the id of the emitting peer
- **emitter** (*str*) – the name the emitter. If None, all capabilities will emit to the receiver

---

Note: A third node can instruct two nodes to subscribe to one another by specifying the ids of the peers. The subscription request is then sent to the emitter node which in turn forwards the subscribtion request to the receiver node.

signal_unsubscribe(*recv_peer*, *receiver*, *emit_peer*, *emitter*)
Unsubscribe a receiver from an emitter

Parameters

- **recv_peer** (*uuid*) – the id of the receiving peer
- **receiver** (*str*) – the name of the receiving variable, or None if no receiver was specified when subscribing.
- **emit_peer** (*uuid*) – the id of the emitting peer
- **emitter** (*str*) – the name the emitter, or None if no emitter was specified during subscription

Note: A third node can instruct two nodes to unsubscribe from one another by specifying the ids of the peers. The subscription request is then sent to the emitter node which in turn forwards the subscribtion request to the receiver node.

emit_signal(*emitter*, *value*)
Update the value of the emitter and signal all subscribed receivers

Parameters

- **emitter** (*str*) – name of the emitting variable
- **value** – the new value

on_peer_enter(*peer*, *name*, **args*, ***kwargs*)
This method is called when a new peer is discovered

Parameters

- **peer** (*uuid*) – the id of the new peer
- **name** (*str*) – the name of the new peer

on_peer_exit(*peer*, *name*, **args*, ***kwargs*)
This method is called when a peer is exiting

Parameters

- **peer** (*uuid*) – the id of the exiting peer
- **name** (*str*) – the name of the exiting peer

on_peer_join(*peer*, *name*, *grp*, **args*, ***kwargs*)
This method is called when a peer is joining a group

Parameters

- **peer** (*uuid*) – the id of the joining peer
- **name** (*str*) – the name of the joining peer
- **grp** (*str*) – the name of the group the peer is joining

on_peer_leave(*peer*, *name*, *grp*, **args*, ***kwargs*)
This method is called when a peer is leaving a group

Parameters

- **peer** (*uuid*) – the id of the leaving peer
- **name** (*str*) – the name of the leaving peer

- **grp** (*str*) – the name of the group the peer is leaving

on_peer_whisper(*peer*, *name*, *data*, **args*, ***kwargs*)
This method is called when a peer is whispering

Parameters

- **peer** (*uuid*) – the id of the whispering peer
- **name** (*str*) – the name of the whispering peer
- **data** – the data the peer is whispering

on_peer_shout(*peer*, *name*, *grp*, *data*, **args*, ***kwargs*)
This method is called when a peer is shouting

Parameters

- **peer** (*uuid*) – the id of the shouting peer
- **name** (*str*) – the name of the shouting peer
- **grp** (*str*) – the name of the group the peer is shouting in
- **data** – the data the peer is shouting

on_peer_modified(*peer*, *name*, *data*, **args*, ***kwargs*)
Called when a peer signals that its capability tree is modified.

Parameters

- **peer** (*uuid*) – the id of the shouting peer
- **name** (*str*) – the name of the shouting peer
- **data** (*dict*) – changed data, formatted as a partial capability dictionary, containing only the changed part(s) of the capability tree of the node

on_peer_subscribed(*peer*, *name*, *data*, **args*, ***kwargs*)
Called when a peer subscribes to an emitter on this node.

Parameters

- **peer** (*uuid*) – the id of the shouting peer
- **name** (*str*) – the name of the shouting peer
- **data** (*list*) – changed data, formatted as [emitter, receiver] emitter: name of the emitter on this node receiver: name of the receiver on the subscriber

on_peer_unsubscribed(*peer*, *name*, *data*, **args*, ***kwargs*)
Called when a peer unsubscribes from an emitter on this node.

Parameters

- **peer** (*uuid*) – the id of the shouting peer
- **name** (*str*) – the name of the shouting peer
- **data** (*list*) – changed data, formatted as [emitter, receiver] emitter: name of the emitter on this node receiver: name of the receiver on the subscriber

on_peer_signaled(*peer*, *name*, *data*, **args*, ***kwargs*)
Called when a peer signals that some of its data is modified.

Parameters

- **peer** (*uuid*) – the id of the shouting peer

- **name** (*str*) – the name of the shouting peer
- **data** (*list*) – changed data, formatted as [emitter, value, [sensors1, ...]] emitter: name of the emitter on the subscribee value: value of the emitter [sensor1,...]: list of names of sensors on the subscriber receiving the signal

on_modified(*peer*, *name*, *data*, **args*, ***kwargs*)
Called when some data is modified on this node.

Parameters

- **peer** (*uuid*) – the id of the shouting peer
- **name** (*str*) – the name of the shouting peer
- **data** (*dict*) – changed data, formatted as a partial capability dictionary, containing only the changed part(s) of the capability tree of the node

run_once(*timeout=None*)
Run one iteration of getting ZOCP events

If timeout is None it will block until an event has been received. If 0 it will return instantly

The timeout is in milliseconds

run(*timeout=None*)
Run the ZOCP loop indefinitely

CHAPTER 8

Indices and tables

- *genindex*
- *modindex*
- *search*

S

S

T

U

Z