Swarm Visualiser - COS 301 Main Project Architecture Requirements and Software Architecture

Team: Dragon Brain
Members:
Matheu Botha u14284104
Renton McInytre u14312710
Emilio Singh u14006512
Gerard van Wyk u14101263

Contents

1 Vision of System

The vision of the system, as expressed by the client, would be to create a standalone, fully functioning, experimental and teaching tool that brings to life, the functioning of a Particle Swarm Optimisation problem solver coupled to a real time graphical visualiser to display the workings of the Particle Swarm Optimisation problem solver to the user.

2 Scope of System

The Swarm Visualiser, as commissioned by Mr Christopher Cleghorn, has two fundamental responsibilities that are encapsulated within a single software program that is deployed to and used from a single computer at a time.

The first responsibility is to provide a underlying, adaptable and fully functioning Swarm Based Optimisation System that makes use of Swarm Based Optimisation algorithms such as the Particle Swarm Optimisation Model or PSO Model to solve problems. The problems that need to be solve are mathematical functions, of various dimensionality and domain.

The second responsibility, and the more important one, is to provide a real time graphical visualisation of the Swarm Based Optimisation Algorithm as it functions in terms of visualising all essential elements of the Swarm Based Optimisation Algorithm as it performs problem solving and then presenting this information to the user in a real time and understandable manner.

To this end, our system is ultimately responsible for providing both the underlying infrastructure in which the Swarm Based Optimisation Algorithm will operate but also for providing the interface infrastructure through which the user will access the underlying Swarm Based Optimisation Algorithm functionality.

2.1 Extended Scope

Below are some feature that extend on the core scope which would enhance the systems usability. The stretch goals are listed below.

- Functionality for a use to generate a predefined starting population for an algorithm in the form of a csv file and import it into the system.
- Functionality to export the results of an algorithm to a csv file.

3 Software Architecture Overview

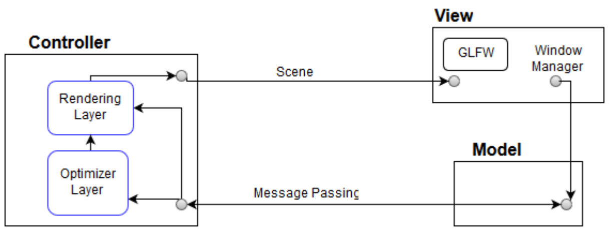

Figure 1: A diagram to indicate the proposed system architecture

4 Architectural Requirements

4.1 Scope

In terms of the Architectural Scope of the project, we have a task that requires a minimal reliance on extremal frameworks and APIs. This is on account of the fact that our product is at its core a desktop application designed to be run in an isolated environment. Additionally, the focus on minimal interference requires that we design the system in such a manner that there is as low a possibility of bottlenecking as possible. As such, we will (as far as possible) minimize the technologies being used to standard C++ and OpenGL. Additionally, the system must be designed in such a manner that the Visualizer and the underlying Swarm Based Optimisation Algorithms are not tightly coupled. It should be easy to adjust one or the other without making adjustments throughout.

4.2 Quality Requirements

Performance Performance is arguably the most vital requirement in the system that in the Visualizer's functionality. It can be defined as follows:
The amount of work accomplished in a measured time interval.
In our case, we are going to make use of reference to **latency** and **frames per second** as a measure of performance, where latency is defined as *a time interval during which a response is achieved given some request* and frames per second, or fps, is defined as *a measurement of how many unique consecutive images can be shown in a graphical context per second.* It should be noted that due to the graphical implications of our task, an important factor in performance is rendering resolution.

As such, the following requirements are set:

- The architecture of the design should be efficiently designed such that when a request is issued to the Visualizer, such as changing the objective function, there is a minimal latency involved in generating a response.
- The visualizer should be allow at least the following resolutions: 800x600, 1024x789, 1920x1080
- Given the maximum resolution 1920x1080, the Visualizer should be capable of running at a consistent 60fps or greater when being run using a mid-tier or above Graphical Processing Unit (for example).
- Given the fact that split-screen functionality is to be implemented, this must be done in a manner such that performance is not hindered dramatically (hence, do not render multiple full resolution images and shrink them post-rendering).

Scalability Scalability refers to the project's ability to handle a large workload for extended time periods and the methods through which this is dealt. Being an isolated application, the only workloads to be experienced are in terms of internal utilization. Hence, in terms of the task described, potential scalability issues lie within the task of assigning a large number of particles in a particular instance of running.
However, the system must still be capable of handling a large number of particles in order to achieve a satisfactory result. As such, the requirement in place is that the system should be capable of handling a large (with some upper limit based on hardware limitations) number of particles while still obtaining a good performance result. This will be achieved through effective use of object memory management and design patterns.

Flexibility Flexibility is a very important quality requirement for the system. Various pieces of the underlying PSO or Particle Swarm Optimisation system that the Swarm Visualiser is meant to be visualising are dependent on configuration parameters and pluggable components such as Objective Functions. It is important that the system be flexible enough so that users can modify the operational parameters of the system without having to perform major code changes, ideally without having to perform code changes at all.

This is particularly important when you consider that the system should not depend heavily on hard-coded components like objective functions but must rather be able to switch to new ones as and when the client requires it.

Usability Usability is another important quality requirement. Ultimately, the system is meant to be used in some capacity as both an experimental tool and a tool for aiding in teaching. Both of these would therefore require that the system be fundamentally easy to use. Some degree of technical competence is assumed on the part of the users,but for the most part, interfacing with the system to access core functionality should not be unintuitive or frustrating.

An additional aspect of this usability is defined in terms of how usable the interface to perform modifications to the system must be. The system, as envisioned, is meant to be highly configurable and the means by which this is accomplished must be as simple as possible.

Although not strictly mandated, traditional values of usability design will be considered in order to deliver on a user-friendly interface that supports maximum usability without sacrificing functionality.

4.3 Integration and Access Channels

The Swarm Visualiser or Particle Swarm Visualisation System that we are developing has very little in the way of integration requirements in terms of needing to interface and integrate with external services and programs. Rather, the client has expressed a desire for the system to function largely as a self-dependent and standalone system meaning that all of the functionality required by the system will be provided on-site by the program.

That being said, the requirements for the system in terms of access channels is much more important. The client has specified that they wish for a single-point-of-access system. This translates to the provision of a single user interface that the client(s) will use to interact with and express the system's functionality. This single user interface must be designed such that it provides the graphical/visualisation requirements as specified as by the client, such as 4-screen display and support for various screen resolutions, but also must provide an interface through which the client can interact with and access the underlying system.

Further provision for alternative deployment systems, beyond deployment of a single computer-based application, is limited due to the existing hardware demands that are contiguous to the performance requirements of the system.

5 Architecture Patterns

5.1 Variant MVC with Layered Controller

Motivations

- We separate the Graphical engine from the optimisation algorithm.
- We enable concurrent development of system components as each component is largely independent.
- It enables different team members to work on different components without being dependent on the progress of the other components.
- It enables components of the system to be modified and updated without requiring system wide updates.

Components The Model View Controller design is structured as follows: The Model handles business logic and data, the View presents data to the user as some valid interface and the Controller receives requests and calls appropriate resources to handle them.

- Model: The Model in our system here would be a shared data storage pool that would be used by the optimiser to push data to as it performs its operations. Furthermore, the Rendering layer will be polling, or pulling data, from the model to satisfy rendering demands. The great benefit of this is that because both components can perform pushes and pulls to the shared data source, and synchronisation can be applied to ensure data integrity, we can decouple the Rendering Layer from the Optimiser layer and cater for various differences such as latency or frames per second. The rendering layer will always render what it can get from the Model at the point of call, and the Optimiser layer will always push what it has currently done. From this, we can apply various synchronisation points that allow us to stagger/bridge the gaps between both layers.

- View: The View in our system is representative of the Client or Interface Layer (often referred to as the Client component throughout this text). The user will be using this layer to view the results of visualisations but it also serves as the primary input point for configuring the Optimiser and the Rendering Layers.

- Layered Controller: The Controller in our system is split into two subsections designed in a layered architecture (bound to the local scope). The two sections are the Rendering Layer and the Optimiser layer.

    1. Optimiser Layer: This is the sub-component of the system in which the optimization algorithms are realized. In short, a general purpose interface for an optimization algorithm is defined which is then realized. This section then performs the function of a generic interchangeable optimization algorithm while updating the shared data pool as it operates. The optimiser will then notify the second layer, the Rendering Layer, when some job can be done.
    2. Rendering Layer: This layer is responsible for the actual visualisation of the system. It makes use of the shared data storage pool and, when it receives some notification (when some changes have occured to the data pool) from the Optimiser layer, it renders the scene accordingly and passes that back to the View, to end the MVC cycle.

5.2 Event-Driven Architecture

The above mentioned MVC pattern highlights and specifies the system components at a high level. This high level description does not encapsulate a behavioural specification at an architectural level in terms of the project. To this end, we turn towards the Event-Driven Architectural pattern to further specify the system.

To begin to understand the relevance of the pattern to the system, consider the following definitions:

- Event: A change of State.
- Event Flow: a sequence of layers that mediates the event in the system.
  - Event Generator: The Event Generator is the component that produces the initial event and the initial sense in the system that an event has occurred.
  - Event Channel: The Event Channel is the medium by which the event is processed to the processing engine.
  - Event Processing Engine: The Event Processing engine is where the event is selected and the response is executed.
  - Downstream Event-driven Activity: Here is where the consequences of the event being processed are made visible.

Considering the above definitions, we will consider a use case of the system to indicate how the Event-Driven Architectural pattern is in play.

As shown by the diagram below, a standard use case of system operation would be the running of an iteration of a Optimisation Process. The Iteration involves 3 components: an Optimiser, a Datastore(Snapshot Manager) and a Graphics Processor.

The Optimiser at the point of 1, has finished perform an iteration. The end result of the iteration, is producing a snapshot. This results in our Event being created. The Snapshot Manager is the communication channel for this event. The enqueue of the Snapshot makes that event visible to the Event Processing Engine, the Graphics Processor. The Graphics Processor is made aware of the new system state once the enqueue is complete and it will take the appropriate action, by dequeuing the snapshot and then

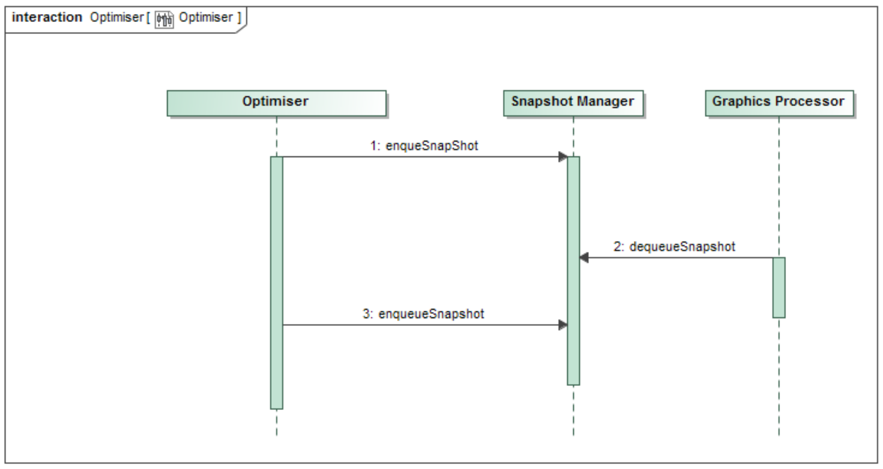

Figure 2: A sequence diagram to indicate the point of events in the system

processing it. Once this is complete, the Snapshot Manager will have one more space, which is the consequence propagated to the system. The cycle begins again with the Optimiser calling enqueueSnapshot again.

This event-driven cycle is found in many other areas of the system since the various components will only react once other components have started to function, furthering the system by processing and generating events, which can be defined at various levels of system granularity.

6 Programming Paradigm

In the programming Paradigm section, we will present a paradigm that we have considered for the project and used in order to adapt to a more traditional imperative paradigm as dictated by the current language implementations of C++.

6.1 Functional Reactive Programming

Functional Reactive Programming is one such programming style that promotes data driven design. There are 3 common divisions of FRP:

- Discrete - Formulations such as Event-Driven FRP and Elm require that updates are discrete and event-driven. These formulations have pushed for practical FRP, focusing on semantics that have a simple API that can be implemented efficiently in a setting such as robotics or in a web-browser.

  In these formulations, it is common that the ideas of behaviors and events are combined into signals that always have a current value, but change discretely.

- Continuous - The earliest formulation of FRP used continuous semantics, aiming to abstract over many operational details that are not important to the meaning of a program. The key properties of this formulation are:

  - Modeling values that vary over continuous time, called "behaviors" and later "signals".
  - Modeling "events" which have occurrences at discrete points in time.
  - The system can be changed in response to events, generally termed "switching."
  - The separation of evaluation details such as sampling rate from the reactive model.

  This semantic model of FRP in side-effect free languages is typically in terms of continuous functions, and typically over time.

There are two common usages of FRP in systems: push-based and pull-based systems. Both have potential advantages and disadvantages within the context of the system.

- Push-based systems take events and push them through a signal network to achieve a result. This would be then that events created by the optimiser layer would then be pushed through the signal network such that they were delivered to the client component via the Rendering layer.

- Pull-based systems wait until the result is demanded, and work backwards through the network to retrieve the value demanded. This would be the opposite in that the client component would demand certain results/services and the request would be filtered down towards the optimiser layer and then delivered back to the Client component via the Rendering layer.

By making use of a shared data pool, to which both the Rendering and Optimiser layer are able to pull and push data from, we obviate the need for an explicit message passing framework.

6.2 Adapations for Project

6.2.1 Language Support

The most pressing barrier for adopting this programming paradigm directly into our project would be the lack of existing C++ implementations for it. The language choice for the project was supplied by the client and this has meant direct adoption of the paradigm is not possible within operating parameters.

6.2.2 Implications for the Project

Despite a lack of direct adoption, we have been strongly influenced by the paradigm with regards to designing the project to make use of the methods and systems espoused by Functional Reactive Programming. Although following an Imperative paradigm, we have structured the system using both events and MVC to create a system with push/pull-based systems, events and discrete and continuous mathematical processing.

7 Unit Testing

With a data orientated approach, your aim is to create a compartment of code that contains functions that will typically only have a single use. i.e. no decisions will ever be made in the functions, the decisions will be made the the graphics engine and the graphics engine will use this grab bag of functions to execute whatever it has decided. This approach allows for powerful, comprehensive unit testing due to the fact that the functions are minimalistic in nature. This is especially useful when debugging a graphical applications as you can unit test an image output, but if you have a comprehensive unit testing backbone for the functions then it can make it easier to see logical mistakes.

The basic premise of data orientated design is to design the code around the data, instead of abstracting the data behind models. While this may seem counter intuitive it is still possible to achieve a familiar level of abstraction while still making use of the performance increases that come with the approach.

At an architectural level, unit testing is typically not present. However, the specific testing framework, Google Test, requires a specific file directory hierarchy and structure in order to work. This requires us to alter directory structure and makelist files such that a Google Test folder can be injected into any working branch of the project so that unit tests can be run without having to perform a manual restructuring of the project.

7.1 Client Layer

The client layer will handle Graphical User Interface elements that the software user will interact with, and will present the output of the Graphics layer(and implicitly the Optimiser layer) to the user. The user will also use the client layer to set parameters of the optimiser layer to determine which fitness landscape will be used, which optimisation algorithm will be used, and also let the user set the optimiser's parameters.

7.2 Technologies

7.2.1 OpenGL

OpenGL was chosen as the graphics engine due to the fact that it will work cross-platform, as opposed to DirectX, and many of the CIRG members run Linux machines, as per client stipulations. Since this program is primarily going to be used by them it makes sense to cater for the target market.

7.2.2 Fruit

Fruit is a dependency injection framework for C++, loosely inspired by the Guice framework for Java. It uses C++ metaprogramming together with some new C++11 features to detect most injection problems at compile-time. It allows to split the implementation code in "components" (aka modules) that can be assembled to form other components. From a component with no requirements it's then possible to create an injector, that provides an instance of the interfaces exposed by the component. This is a good option for the DI framework as unlike other c++ DI frameworks most of the checks are done at compile-time to try catch the errors early. Another bonus is that the syntax is similar to jUnit so it will be more comfortable to work with initially.

7.2.3 GLEW

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file. GLEW has been tested on a variety of operating systems, including Windows, Linux, Mac OS X, FreeBSD, Irix, and Solaris.

7.2.4 GLM

The visualiser component will make use of the OpenGL Mathematics(GLM) framework. This is a light-weight, optimized mathematics framework that handles matrix manipulations and shader control.

7.2.5 GLFW

The interface will make use of the GLFW. GLFW is an Open Source, multi-platform library for creating windows with OpenGL contexts and receiving input and events. It is easy to integrate into existing applications and does not lay claim to the main loop.

7.2.6 CMake

CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files.

7.2.7 Fraps

Fraps is a universal Windows application that can be used with games using DirectX or OpenGL graphic technology. For our purposes it will be used as the benchmarking software as it can how how many Frames Per Second (FPS) you are getting in a corner of your screen. Perform custom benchmarks and measure the frame rate between any two points. It can also save the statistics out to disk and use them for your own reviews and applications. It also has useful Screen Capture Software as well as Realtime Video Capture Software which can be useful for demoing purposes.

7.2.8 Cppcheck

Cppcheck an open source code analysis tool specifically for the C language. In terms of applicability, Cppcheck will be used to analyse the C++ code written for the project to ensure that a single consistent coding standard that will add in the maintenance of the project as well as provide a consistent standard that can be enforced across all of the system documentation.

8 Architectural Components

8.1 Graphics Processor

The graphics processor, as a component, is responsible for converting the raw data generated by the optimiser and visualising it in a user friendly manner.

8.1.1 Performance

With the requirement that the display should maintain 60fps under certain hardware conditions the performance of the Graphics processor is of key importance. Most of the heavy duty work that the processor will be doing will be in the shaders which are notoriously difficult to debug and test as they are written in GLSL, a domain specific language which has no sort of testing framework. Ways to approach this would be to have many simple shaders and then layer those shaders.

8.1.2 Scalability

The graphics processor needs to be scalable the render capability can be scaled. This can be done simply as the boiler plate code is fairly static and then by simply abstracting the shader program it is simply to simply add/remove and layer the shaders to fully customise the render quality. The graphics processor also needs to be able to visualise multiple instances of optimisers.

8.1.3 Reliability

The graphics processor should be able to recover if an error were to occur in either the message passing or optimiser. Correctly signalling the graphics processor should keep the user informed about what is happening in the system.

8.2 General Optimiser

The General Optimiser is much less performance bound than the other system modules because of the use of an intermediary. Generally speaking, the Graphics Processor will faster or slower than the General Optimiser, depending on a variety of factors. Typically, this means that the General Optimiser can afford to, with the appropriate measures, function at a slower speed than the Graphics Processor.

8.2.1 Quality Requirements

1. Flexibility: The General Optimiser must be flexibility enough in design such that a user parameter configuration change will be handled internally without requiring a program restart or recompile, that is perform runs, and then be able to perform further runs without requiring recompilation of the system.

2. Performance: As mentioned above, the performance requirements of the General Optimiser are not as stringently defined as with the Graphics Processor. However, the requirement is somewhat implicit in the fact that the optimiser will have to provide the graphics processor with enough data to allow it to animate the landscape smoothly but perhaps a more important performance condition would be the stopping conditions. Being able to accurately predict when a swarm has stagnated will allow for the simulation to exit in a timely manner so that a user does not have to wonder whether or not something is happening.

3. Reliability: Reliability is an important concern in that the General Optimiser must be reliable in two ways: the first is the component itself should not fail often. In the event of failures, it must be able to recover without compromising the system. The second area, perhaps more critically, is that the results produced by the General Optimiser in terms of the particle positions and values etc must be consistent with the expected problem solving capabilities and capacities of Swarm-based problem solving methods. That is, we should not expect variance in results that would not be possible of a Swarm-based problem solving approach to produce.

4. Scalability: Scalability is an important concern in that the user has the capacity to perform simultaneous problem solving methods on multiple screens. The General Optimiser must therefore be able to support multiple problem solving runs at the same time.

8.3 Manager

The Manager component is the high level component that is the core component that serves as the intermediary component between all other components in the system. All other components will pass messages to the Manager who will then pass further messages as needed, which establishes and integrates the system as a whole in terms of communication pathways.

8.3.1 Quality Requirements

1. Integrability: The Manager component is the core of the integration efforts.No components directly interact with each other and as a result, all components direct interact with the Manager. This means that the Manager must be Integrable in terms of supporting communications between the other components.
2. Reliability: Reliability is a core component of the Manager module in that it must be reliable in terms of the component failing as rarely as possible. Should the manager component fail, then the system as a whole will suffer a critical error that will not be recoverable.
3. Scalability: The Manager will have to manage potentially multiple concurrent Optimisers and other components operating in concert to solve multiple problems. In this way, the Manager must be able to scale to function with as many as 4 concurrent problem solving and as few as 1.

8.4 User Interface

The purpose of the user interface component is to initially allow the user to create the necessary settings package and then allow the user to run optimisers according to their specifications.

8.4.1 Reliability

The interface is the primary point of information access for the user. In this respect it should be robust and informative to the user.

8.4.2 Usability

The interface should conform to good usability standards in terms of not requiring extensive training and not requiring retraining in order to use it again after sufficiently large time has elapsed between uses.

8.4.3 Scalability

The interface should be able to scale with evolving functional requirements and still maintain good usability.