Software Design and Architecture (SDA)

1. Introduction

This document defines the architecture of an artificial life simulation application. The goal of the project is to develop a robust, high-performance framework within which emergent biological phenomena (e.g., evolution, population dynamics, speciation, etc.) can be observed. Performance, flexibility, and extensibility are the main focuses of the design.

2. Architectural Goals and Principles

Goals:

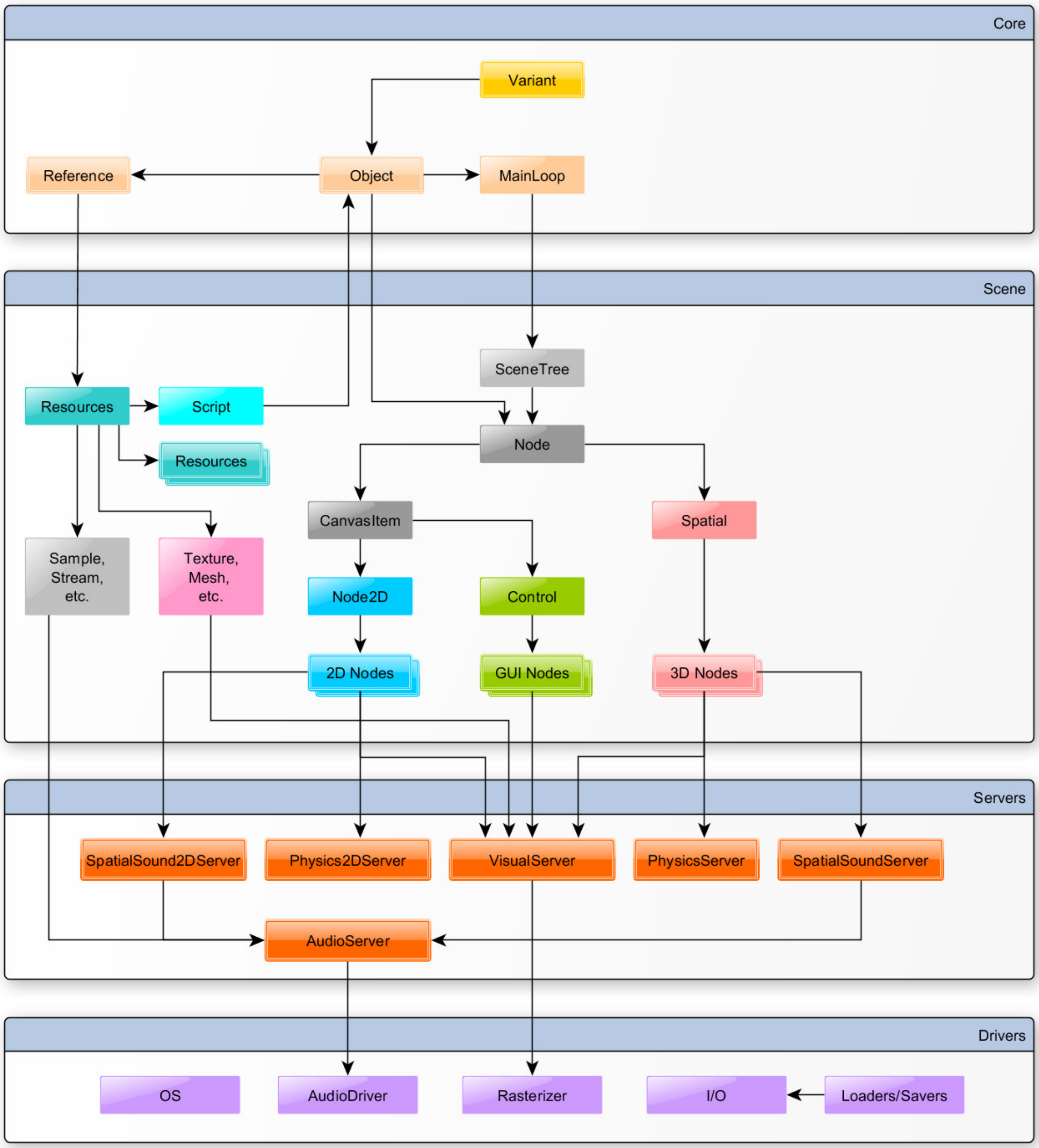
- **Performance**: the software must run smoothly with large amounts of data. I.e., simulations should not encounter stutters/crashes during runtime.
- **Flexibility**: the software must be adaptable to changes throughout the development process.
- **Extensibility**: the software must allow for new features to be easily added

Principles:

- **Modularity**: the software system is logically divided into self-contained components
- **Separation of concerns**: each software element should have a single responsibility
- **Bounded contexts**: each conceptual domain in the system should separately control its own implementation and should interact with other domains through fixed program interfaces

- **Minimal coupling**: software elements should have low interdependence
- **Testability**: software components should be individually and collectively verifiable
- **OCP:** Develop code modules to be open for expansion but closed for modification. I.e., maintain expandability with minimal implications unto source code.

3. System Overview



The Godot 4.2 software architecture diagram (source).

4. Architectural Patterns

The Godot game engine was selected as a framework for this project. As illustrated by the above diagram, it uses a layered architecture with four levels. This design pattern aligns with the product requirements because it allows for separation of the graphics from the simulation itself. Additionally, it leverages concurrency to provide superior performance to the application overall.

5. Component Descriptions

A Godot application has four main components:

- **Core**: The core is the Godot engine itself–the framework within which applications can be developed and run.
- **Scene**: The scene component includes all elements and data for a particular application in Godot.
- **Servers**: The servers are various concurrent processes for running and rendering the application.
- **Drivers**: The drivers are parts of the operating system that allow the application to be run on user devices.

6. Data Management

A Binary Search Tree (BST) structure will be used to track the hierarchy of each node (organism) and maintain efficiency for efficient insertion, deletion and searching. This will ensure the data structure is scalable and efficient for different simulation runs.

7. Interface Definitions

The user will be able to pull up all information on a specific genome and lineage in a readout when the organism is selected on screen. This will rely on data actively updated in the BST during runtime to display statistics related to each organism by presenting the member variables in a visual readout. This means each node will need to be able to be efficiently searched through when prompted to find the correct one and provide the associated information.

A general information readout will be provided to give the user an overview of the simulation environment. This readout will rely on broader information from the BST, such as the total number of nodes (total organisms), the number of nodes on a specific branch (total organisms from a specific lineage), as well as general information such as how much simulated time has passed.

8. Considerations

8.1. Security

The project does not require specified user data, authorization, or authentication and will not be connected to the internet, thus there is a low possibility of security risks.

8.2. Performance

Our program is a simulation that will vary with different parameters and will need to handle a large amount of data. Strategies like load balancing and vertical scaling will be adopted along with the BST to meet high performance and responsiveness requirements, ensuring the simulation will run smoothly even at large scales.

The C++ language is being used for the system and scripting due to its natural performance benefits while being able to represent complex objects. In addition, the framework is split into a graphic frontend applied to a rigorous simulation backend to ensure the backend elements handle efficiency and data management outside visual interactions.

Amongst other strategies, a low level of detail (LOD) will be applied to rendered entities. Bounding box optimization, and parallel processing may be used to ensure smooth runtime performance.

8.3. Maintenance and Support

Our project will be producing a standalone application without a server component, so there won't be system support beyond the application itself. If problems are discovered, issues can be submitted to the GitHub repository that we'll work on and implement fixes for. As features are added we'll periodically have releases, and there will always be a release before a major version change.

As our project will be open source and we don't have a project partner who will be taking over the project upon completion of the capstone, either one of us will become the project maintainer or the project will become unmaintained. There's also the possibility of another capstone team taking over the project after we finish.

9. Deployment Strategy

Our application is going to be compiled into an executable which will be runnable on standard x86 computers.

10. Testing Strategy

As we develop we'll simultaneously write unit tests/integration tests/etc., which will be run as part of our CIT system and be shown in the information about open PRs. All tests will need to be passed for a PR to merge.

Strict stress testing and user acceptance testing will be done within a sample simulation environment to verify that the efficiency is meeting appropriate standards. This includes being able to handle enough generations to see noticeable mutations and a consistent lack of program stuttering and bugs.

User acceptance testing will be done on the UI to verify that the modifiable components provide varying amounts of information appropriate for different user's experience levels. These tests will also extend to verifying the clarity and visual appeal of UI elements.

11. Glossary

Definitions of technical terms and acronyms used in the document:

- API: application programming interface
- CIT: continuous integration testing
- PR: pull request
- BST: binary search tree
- LOD: level of detail